NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

FINDING BENT FUNCTIONS USING GENETIC ALGORITHMS

by

Stuart W. Schneider

September 2009

Thesis Co-Advisors: Jon T. Butler
Pantelimon Stanica

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704–0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704–0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2009	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE Finding Bent Functions Using Genetic Algorithms			**5. FUNDING NUMBERS**
6. AUTHOR(S) Stuart W. Schneider			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943–5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

In this thesis, a generic genetic algorithm (GA) is presented that is implemented on a reconfigurable computer. Our GA is implemented such that many problems can be solved by simply adapting the problem to the GA. For example, part of this process involves the customization of the fitness function of the given problem to the GA. The size of the problem is limited by the capacity of a field programmable gate array that is part of the reconfigurable computer. We apply this to bent functions, which are Boolean functions that are well suited for cryptographical applications and are extremely rare. Experimental results show the effectiveness of this technique. Different methods are used to discover bent functions. These methods take advantage of the properties of bent functions to reduce the total search space. This allows a brute force search to be conducted on the reduced search space to locate the set of bent functions in that search space. Two different methods are used to reduce the search space. The first is through rotationally symmetric functions, which reduces the number of bent function that can be found, while the second is by the degree of the function, which locates all bent functions.

14. SUBJECT TERMS Genetic Algorithm, Bent Functions, Reconfigurable Computer, Field Programmable Gate Array (FPGA), Cryptology			**15. NUMBER OF PAGES** 200
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540–01–280–5500 Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

FINDING BENT FUNCTIONS USING GENETIC ALGORITHMS

Stuart W. Schneider
Lieutenant, United States Navy
B.S.EE and B.S.CS, United States Naval Academy, 2000

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2009

Author: Stuart W. Schneider

Approved by: Jon T. Butler
Thesis Co-Advisor

Pantelimon Stanica
Thesis Co-Advisor

Jeffrey B. Knorr
Chairman, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

In this thesis, a generic genetic algorithm (GA) is presented that is implemented on a reconfigurable computer. Our GA is implemented such that many problems can be solved by simply adapting the problem to the GA. For example, part of this process involves the customization of the fitness function of the given problem to the GA. The size of the problem is limited by the capacity of a field programmable gate array that is part of the reconfigurable computer. We apply this to bent functions, which are Boolean functions that are well suited for cryptographical applications and are extremely rare. Experimental results show the effectiveness of this technique. Different methods are used to discover bent functions. These methods take advantage of the properties of bent functions to reduce the total search space. This allows a brute force search to be conducted on the reduced search space to locate the set of bent functions in that search space. Two different methods are used to reduce the search space. The first is through rotationally symmetric functions, which reduces the number of bent function that can be found, while the second is by the degree of the function, which locates all bent functions.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

In this thesis, several methods are shown to locate bent functions. Bent functions are well suited for cryptographical applications, such as in the substitution box in the DES encryption standard, or the Grain-128 cipher [7]. To the best of our knowledge, this is the first time that a reconfigurable computer has been used to locate bent functions on more than six variables. Due to the repetitive nature of an algorithm needed to determine if a function is bent, reconfigurable computers are ideally suited to locate them, especially when compared to a general purpose computer.

As a result of this thesis, the Naval Postgraduate School (NPS) now has 5,425,430,528 6-variable bent functions, and 1,933,312 8-variable ROTS bent functions for use in additional thesis and research work. Additionally, calculators have been created to allow a nonlinearity calculation to be made on 8- and 10-variable functions. This was not previously possible at NPS due to a lack of memory error that occurs when attempting to complete the nonlinearity calculation using a previous instantiation of the algorithm.

Multiple ways were examined to locate bent functions. Since there is not enough time to conduct a brute force search of all functions on more than four variables, three different means were used to restrict the search space to locate bent functions. The first was by examining rotationally symmetric (ROTS) functions. The second method was to search for them according to the degree of the function as revealed by its algebraic normal form. This is accomplished through sequentially enumerating all of the functions according to their degree through an index. This method revealed an interesting fact that bent functions commonly occur with consecutive indices. Next, a genetic algorithm was used to create a sieve to locate ROTS bent functions. These results show that, through a well-designed chromosome, fitness function, crossover point and minimum fitness value, the ability to locate bent functions is drastically increased. Finally, a non-traditional approach towards GAs is taken to identify bent functions on 4 variables using other 4 variable bent functions.

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

First, I would like to thank LT Jennifer Shafer for explaining the mapping on the transeunt triangle. I would next like to thank my thesis advisors Dr. Jon Butler and Dr. Pantelimon Stanica for all of their guidance and help in making this thesis a reality. I would also like to thank my lovely wife Naoko for all of her patience and long suffering as I worked late into the night. Finally, I would like to thank my Lord and Savior, Jesus the Christ, for all of His help, guidance, and inspiration.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. CRYPTOLOGY APPLICATIONS

Two of the key aspects of a cryptographic system are confusion and diffusion [14], [10]. Diffusion is the process by which repetitive information is "dissipated" over an entire message. A simple case of confusion would be to replace one letter, 'E', with another letter, 'K'. This method does little to improve the secrecy of the message since the frequency graph of the letters has not changed. In order to make this method more practical, several substitutions must be made [14].

In a paper frequently described as "small" and "beautiful," Rothaus introduces a new type of function known as a bent function [5], [6], [13]. The term "bent" was probably chosen by Rothaus because it suggests the opposite of "linear" [3]. These functions are most notable because they have the highest nonlinearity among all functions on the same number of variables. Because of this, they are well suited for cryptographical applications, including being used as part of the substitution box in the DES encryption, or the Grain-128 cipher, to mention just a few [7].

B. GENETIC ALGORITHMS (GA)

Several problems that need to be solved in a business environment involve a cost-benefit analysis. As various scenarios stress the variables differently, desirable components might exist in different solutions. GAs take advantage of two solutions yielding good answers by merging the parameters of the two solutions. Although the groundwork on GA started in the 1950s, the interaction between a conventional microprocessor and a reconfigurable computer is allowing GAs to enter a new realm of truly parallel operations. Although a GA has been implemented on an FPGA, this thesis will examine their implementation in a more parallel manner [19].

C. RECONFIGURABLE COMPUTING ON THE SRC-6

The SRC-6 is a microprocessor based computer that contains additional boards known as Multi-Adaptive Processing (MAPs). Each MAP contains three Xilinx Field Programmable Gate Arrays (FPGAs), two running user programs, and the third controlling the others [16]. To understand how this project helps to locate bent functions, it is necessary to know how the SRC-6 operates. The SRC-6 operates via a dual core general purpose CPU communicating with the FPGAs. There are two methods in which the FPGAs can be programmed to solve a problem. The first is through a hardware description language (HDL), either Verilog or VHDL, and the other is through the high level languages C or FORTRAN. The remainder of this paper will focus on programming the SRC-6 with Verilog and C. Regardless of whether the FPGA is programmed in C or HDL, where the problem solution design is written, the interface between the microprocessors and the FPGA is through C code. Figure 1 demonstrates the interaction between the various components of the SRC-6.

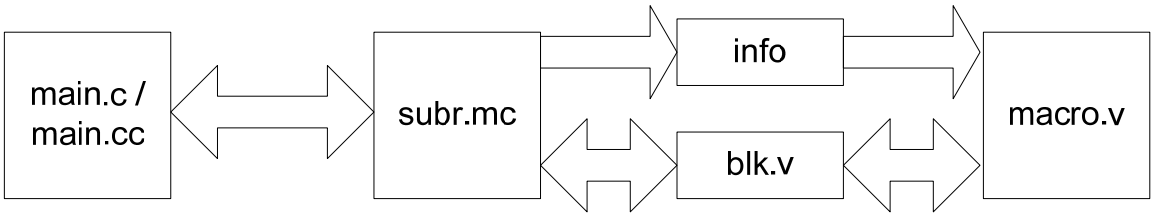

Figure 1 SRC-6 data flow path

The programmer is able to access the FPGAs on the SRC-6 by first writing C/C++ code that is executed on the general purpose computer. When he wants to have the FPGAs process data, he merely makes a C style function call that invokes the SRC-6. Next, program execution is passed to a subroutine that is written in C. This subroutine, subr.mc, is compiled to operate on the FPGA.

When a problem is solved through Verilog, the programmer designs the circuit as he would for any FPGA. The file describing this circuit is macro.v. This module has two interfaces to the subr.mc C code. The first is a blackbox interface (blk.v) that would be analogous to a function prototype in C. This interface merely restates the module name

and port declarations from the module source code. The other interface (info) includes information about the macro, such as if it is pipelined or stateful. It also specifies its latency and additional control signals that need to be applied to the circuit, such as clear and clock signals. This code is directly parsed to generate the interface between the C code and the Verilog code.

D. GOAL OF THIS THESIS

The goal of this thesis is to determine if GAs are useful in finding bent functions. In the process of doing this, several different methods of looking for bent functions will be examined. These methods restrict the search space to make it possible to enumerate all of the bent functions. In doing so, it is desired to see how these methods can be adapted for use in GAs.

E. THESIS ORGANIZATION

This thesis is organized as follows. Chapter II describes bent functions and how to find them. Chapter III is a discussion on genetic algorithms, and how one is implemented on the SRC-6. Chapter IV is a discussion on how bent functions were found for this thesis. Chapter V is a summary of the results. Appendix A contains additional information on the SRC-6. Appendix B are the lessons learned while conducting the research for this thesis. Finally, Appendix C is the Verilog code for the genetic algorithm.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BENT FUNCTIONS

A. BACKGROUND

Bent functions contain many properties that must be examined. These properties are important since they are used to determine if a function is bent, and provide insight in how to construct a function that might be bent.

1. Definitions

a. Linear or Affine Function

A linear function is the constant 0, or the exclusive OR of one or more variables. An affine function is a linear function, or the complement of a linear function.

b. Nonlinearity (NL)

The nonlinearity of a function is the least number of bits that are required to be changed in order to convert the function into some affine function.

c. Bent Function

A function on an even number of variables is called bent, if it has the maximum nonlinearity among all other functions on the same number of variables.

d. A-class

Suppose f is a bent function and a is an affine function. It has been shown that $g = f \oplus a$, is also bent [5]. Because of this, we say that f and g are in the same A-class.

e. Truth Table (TT)

The truth table of a function f, specifies the value of f for all assignments of values to the variables. For example: the TT of $f_1 = \overline{x}_1\overline{x}_2 + \overline{x}_1 x_2 + x_1\overline{x}_2$ is 1110, where $x_1x_2 =$ 00, 01, 10 and 11 map to 1, 1, 1, and 0, respectively.

f. Algebraic Normal Form (ANF)

The ANF of a function f is the exclusive OR of product terms, where all variables occur uncomplemented. For example, consider the ANF of $f_1 = 1 \oplus x_1x_2$.

g. Degree

The degree of a function is the maximum number of variables that exist in any of its terms as expressed in the ANF. For example, let f and g be functions on 3 variables, x_1, x_2 and x_3, $f = x_1x_2 \oplus x_3$ and $g = x_1x_2x_3 \oplus 1$. Then, f and g have degree 2 and 3, respectively.

h. Co-functions

The TT form of a function f can be considered as a double word $f_{0\to x_i}\ f_{1\to x_i}$ which are f with x_i replaced by 0 and 1 respectively. Each of the words that comprise the TT form of a function is called a co-function. The co-function containing the MSB is referred to as the "high" co-function, and the other co-function is referred to as "low".

2. Properties

a. Rotationally Symmetric (ROTS) Functions

A function is ROTS if $f(a_0, a_1, ...a_{n-2}, a_{n-1}) = f(a_1, a_2, ..., a_{n-1}, a_0)$. The number of bits in an n-variable ROTS function is R [5].

b. Maximum Nonlinearity

The maximum nonlinearity of an *n*-variable function $NL_{\max}(n) = 2^{n-1} - 2^{\frac{n}{2}-1}$ [13].

c. Weight

The weight W_f of a function f is the number of ones in its truth table. A bent function f has weight $W_f(n) = 2^{n-1} \pm 2^{\frac{n}{2}-1}$ [13].

d. Summary

Table 1 provides a listing of the properties on 4, 6, 8 and 10 variables.

n	2^n	R	2^{n+1}	$NL_{\max}(n)$	W_f
4	16	6	32	6	8 ± 2
6	64	14	128	28	32 ± 4
8	256	36	512	120	128 ± 8
10	1024	108	2048	496	512 ± 16

Table 1. Bent functions properties by number of variables

B. REPRESENTATIONS

1. Truth Table

Consider Table 2 for the expression $\overline{x}_2$ on three variables. It has a truth table representation of $0x33$. The MSB corresponds to the table entry of, $x_1x_2x_3 = 111$, while the least significant bit (LSB) entry corresponds to entry $x_1x_2x_3 = 000$. The hexadecimal representation of $00110011_2 = 0x33$ has the MSB (Most Significant Bit) being written as the leftmost bit.

$x_1x_2x_3$	$\overline{x}_2$ or $x_2 \oplus 1$
000	1 (LSB)
001	1
010	0
011	0
100	1
101	1
110	0
111	0 (MSB)

Table 2. Truth table

2. Algebraic Normal Form

A function is expressed in its algebraic normal form (ANF) by:

$$f = c_0 1 \oplus c_1 x_3 \oplus c_2 x_2 \oplus c_3 x_2 x_3 \oplus c_4 x_1 \oplus c_5 x_1 x_3 \oplus c_6 x_1 x_2 \oplus c_7 x_1 x_2 x_3$$

Where $c_0, c_1, ..., c_7$ are the values from the ANF table where c_7 corresponds to the MSB. Consider the expression $x_1 x_2 \oplus x_3$ in Table 3. When it is represented in its ANF, its expression is $01000010_2 = 0x42$. Again, the left most bit in the hexadecimal notation is the MSB. However, this time the entry 1 or 0 in the truth table corresponds to whether that term exists or not in the expression. For example, the expression $x_1 x_2 \oplus x_3$ has two terms, $x_1 x_2$ and x_3. Thus, the entries 110 and 001 have a one in their associated column to signify that the term exists in the expression. The ANF is a useful representation, because it can be used to identify bent functions of the same A-class. If two bent functions are of the same A-class, then all of their non-linear term coefficients will be the same.

Term	$x_1x_2x_3$	$x_1x_2 \oplus x_3$ or $\overline{x}_1x_3 + \overline{x}_2x_3 + x_1x_2\overline{x}_3$
$c_0 1$	000	0 (LSB)
c_1x_3	001	1
c_2x_2	010	0
$c_3x_2x_3$	011	0
c_4x_1	100	0
$c_5x_1x_3$	101	0
$c_6x_1x_2$	110	1
$c_7x_1x_2x_3$	111	0 (MSB)

Table 3. ANF table

3. Transeunt Triangle

The transeunt triangle is a data structure that allows the conversion from the truth table form to the ANF, and vice versa [18], [2]. Regardless of the mode of operation, the transeunt triangle receives and processes its data in the same manner. The data for the current format is placed along the bottom row of the triangle, with the MSB being the right most bit. The bits on the next higher row are created by the exclusive ORing of the adjacent bits in the row below it. The ordering of these bits corresponds to the truth or ANF table shown in Table 3 being rotated counter-clockwise 90°. The output table is read along the left side of the triangle. Its corresponding values would be as if the above tables are rotated 120° counter-clockwise. Figures 2, 3 and 4 show the transeunt triangles for Tables 2 and 3.

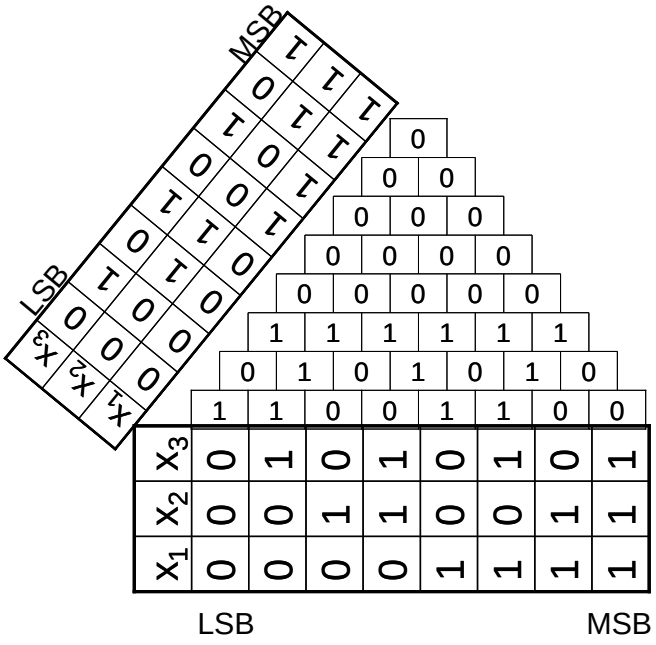

Figure 2 Transeunt triangle conversion for $\overline{x}_2$ to $x_2 \oplus 1$

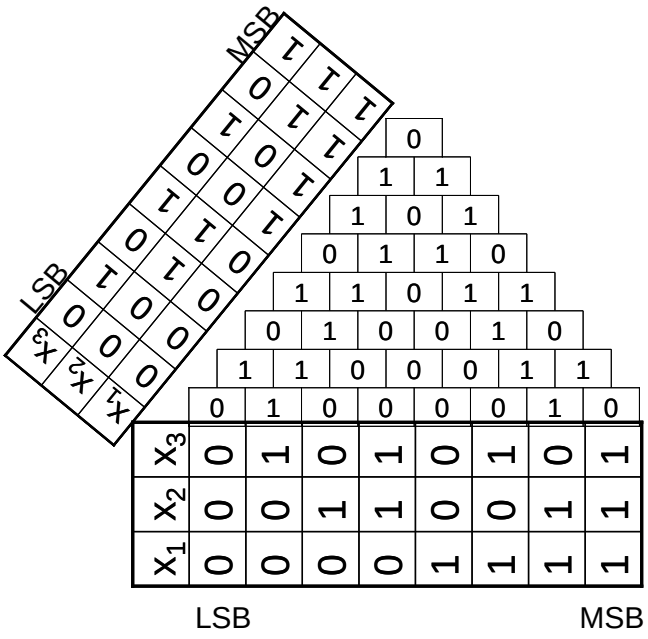

Figure 3 Transeunt triangle for $x_1 x_2 \oplus x_3$ to $\overline{x}_1 x_3 + \overline{x}_2 x_3 + x_1 x_2 \overline{x}_3$

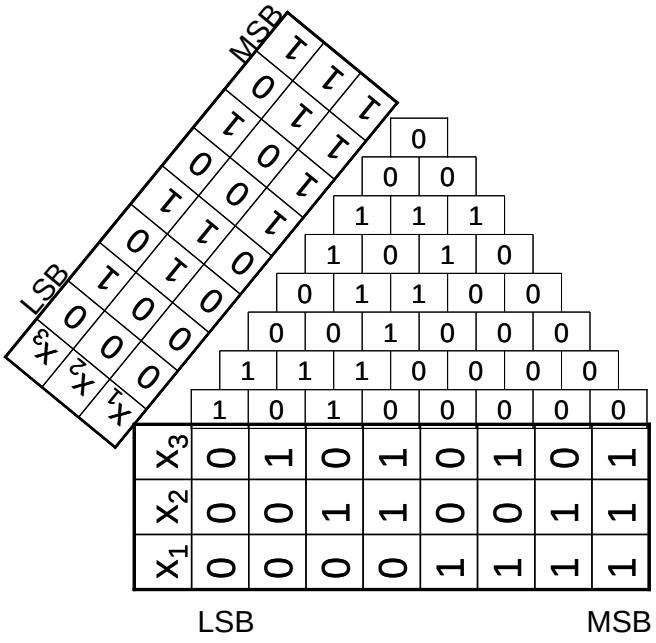

Figure 4 Transeunt triangle for $x_2 \oplus 1$ to $\overline{x}_2$

The transeunt triangle shown in Figure 4 is the inverse of the Figure 2 . On initial observation the truth table expression would be $\overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1\overline{x}_2x_3 + x_1\overline{x}_2\overline{x}_3 + x_1\overline{x}_2x_3$. This, however, can be easily simplified as in Figure 5.

$$\overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1\overline{x}_2x_3 + x_1\overline{x}_2\overline{x}_3 + x_1\overline{x}_2x_3 =$$
$$\overline{x}_1\overline{x}_2(\overline{x}_3 + x_3) + x_1\overline{x}_2(\overline{x}_3 + x_3) =$$
$$\overline{x}_1\overline{x}_2 + x_1\overline{x}_2 =$$
$$\overline{x}_2(\overline{x}_1 + x_1) =$$
$$\overline{x}_2$$

Figure 5 Truth table simplification

C. BENT FUNCTION DISCOVERY

There are several different approaches that can be used to discover bent functions. Regardless of the tool used to restrict the search space to a manageable size, this research uses the same algorithm to determine if a function is bent.

1. General Case

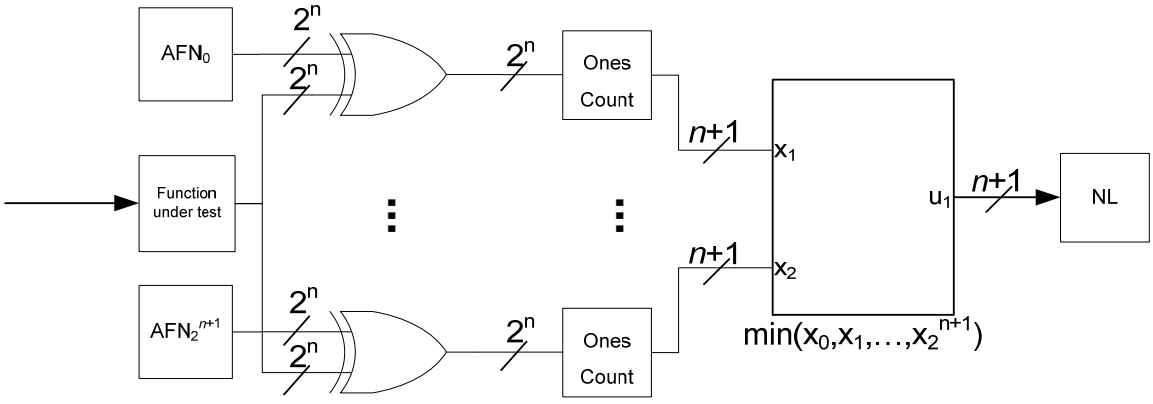

Figure 6 General case nonlinearity calculation

Figure 6 shows a general algorithm to determine the nonlinearity of a function. The function under test is exclusive ORed with each of the affine functions, and the number of ones in each function is counted. The minimum number of ones is then determined over all of the 2^{n+1} calculations which correspond to each of the affine functions. The resulting number is then compared to the maximum nonlinearity for the given number of variables. If they are the same, then the function is bent.

2. Brute Force

The easiest method to determine which functions are bent is to perform a brute force attack. By doing so, the nonlinearity of all functions in S is calculated. By doing so, one also generated a histogram showing the distribution of the nonlinearities. This is beneficial in that it also verifies the maximum nonlinearity equation, $NL_{\max}(n)$. This method is time consuming for $n < 6$ and impractical for $n \geq 6$.

3. ROTS

It has shown that rotationally symmetric functions are rich in bent functions [5]. This means that by enumerating only the ROTS functions, which is considerably smaller than the total number of functions, bent functions can be more readily discovered. The

first step in this process is to determine all of the ROTS functions. Next, each bit is mapped to an index. Table 4 is the Verilog code that produces the necessary mapping for $n = 4$.

```
assign TT[   0] = RSI[ 0];
assign TT[   1] = RSI[ 1];
assign TT[   2] = RSI[ 1];
assign TT[   3] = RSI[ 2];
assign TT[   4] = RSI[ 1];
assign TT[   5] = RSI[ 3];
assign TT[   6] = RSI[ 2];
assign TT[   7] = RSI[ 4];
assign TT[   8] = RSI[ 1];
assign TT[   9] = RSI[ 2];
assign TT[  10] = RSI[ 3];
assign TT[  11] = RSI[ 4];
assign TT[  12] = RSI[ 2];
assign TT[  13] = RSI[ 4];
assign TT[  14] = RSI[ 4];
assign TT[  15] = RSI[ 5];
```

Table 4. ROTS mapping

From this table, a 6-bit counter can be applied to the rotationally symmetric index (RSI) to generate the truth table representation of the ROTS functions. This representation generates the function under test, which is applied to the general case algorithm to determine the nonlinearity as shown in Figure 7.

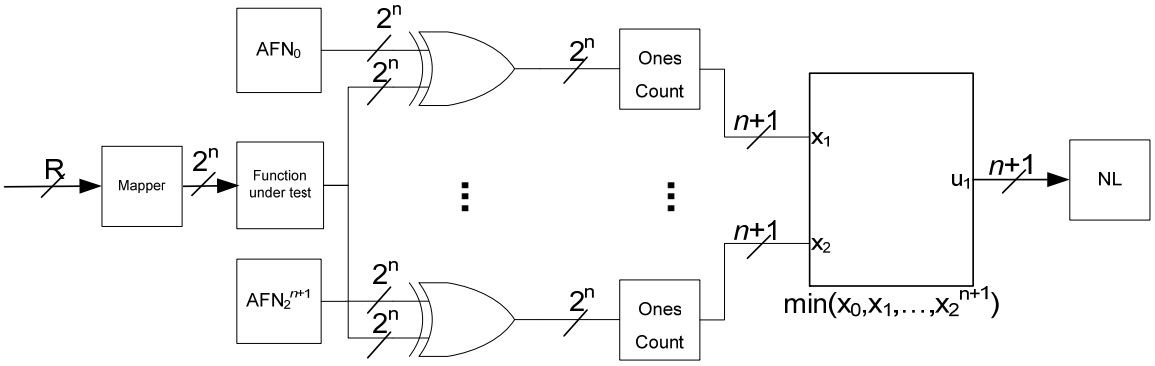

Figure 7 ROTS nonlinearity calculation

4. By Degree

It is known that the highest degree of a bent function is $\frac{n}{2}$ [16]. Furthermore, in the ANF of a function, the linear terms can be ignored, since they only differentiate two bent functions in the same A-class. This, significantly reduces the total search space to terms of degree 2, 3, …, and $\frac{n}{2}$. For the case of $n=6$, the highest degree of a bent function is 3. This means that the only functions that must be enumerated are those of degree 3 and 2. Thus, the total number of bits in the search space is now $\binom{6}{2}+\binom{6}{3}=15+20=35$. This results in only $1.86x10^{-7}\%$ of the original search space of 2^{64} functions, if the search is exhaustive. Figure 8 shows the original algorithm as modified to accomplish this.

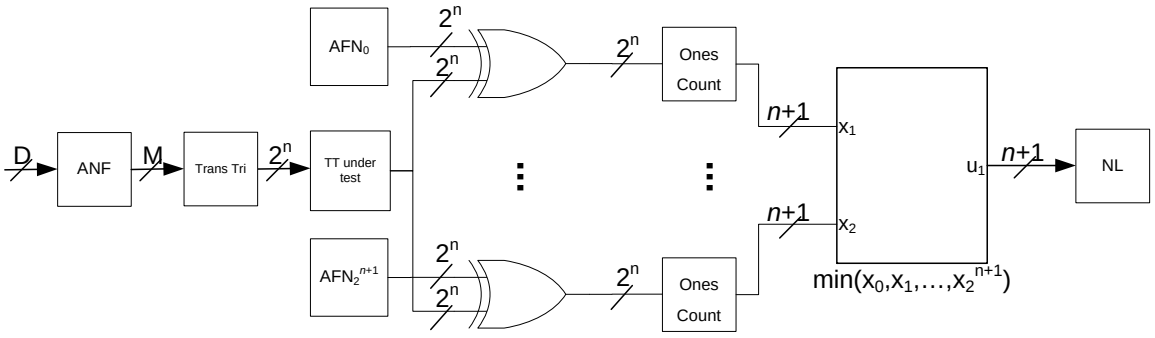

Figure 8 Nonlinearity calculation by degree

5. Complement Optimization

The number of affine functions that need to be evaluated can be cut by ½. This is done by recognizing the relationship between nonlinearity of a function with respect to a particular affine function, and that affine function's complement, and will be shown by example. For any given n variable bent function, there are 2^{n+1} affine functions. However, because of the linear nature of the affine function, it is only required to enumerate 2^n of them since their complement will yield the other 2^n affine functions. This property relationship can be applied to the nonlinearity of a function.

Consider the case when $n = 4$, and the affine functions $0x0000$ and its complement $0xFFFF$. Consider a function $f = 0xC3D7$. Its nonlinearity with respect to $0x0000$ is 10, and its nonlinearity with respect to $0xFFFF$ is 6. Recalling that there are 16 bits in each function on four variables, and given the affine function $0x0000$, we can determine the nonlinearity of its complement affine function, $0xFFFF$, by subtracting its nonlinearity from 16. For this case, $16 - 10 = 6$. In general, the below formula can be utilized to determine the minimum nonlinearity, $NL_{\min}$, of a function given the NL one of the affine functions.

$$NL_{\min} = \min(NL, 2^{n+1} - NL)$$

Through this calculation, the following effects are observed on the circuitry required to implement the algorithm. The number of exclusive OR gates required is reduced by ½, along with the number of ones count calculations. However, the number of minimization calculations required is constant, because an additional minimization is implemented with the aforementioned subtraction. 2^n subtraction units must be added to the $NL_{\min}$ calculation. However, the complexity of that operation is insignificant compared to the required circuitry to implement the ones count algorithm for the affine functions not directly tested.

D. SUMMARY

This chapter describes bent functions, and introduces their various properties. It is through application of these properties that locating bent functions is possible, given their rarity. The next chapter discusses genetic algorithms in preparation for how to locate bent functions.

THIS PAGE INTENTIONALLY LEFT BLANK

III. GENETIC ALGORITHMS

A. BACKGROUND

Genetic algorithms have their basis by what is seen in nature. The three main processes that will be discussed are survival of the fittest, crossover and mutation. Survival of the fittest is similar to the idea that the stronger animals in a herd are more likely to live and go on to produce children for the next generation. Crossover is based on the possibility that if two parents have desirable traits, their children may have a combination of those desirable traits. Finally, mutation is the idea that a change in a gene might make the animal more resilient for its environment.

1. Definitions

a. Chromosome, Element or Member

A potential solution to a problem is encoded as a string and is referred to as the chromosome. Strings can be any combination of characters, but this thesis will only consider the case of binary digits. A chromosome may also be referred to as an element.

b. Gene

In this thesis, a gene is each character of the chromosome.

c. Value

The value is a numerical representation of the chromosome. This can be created in any number of methods. One common method is to directly convert the chromosome into its integer representation. In a genetic algorithm involving trigonometric functions, for example, the value could represent some fraction between $-\pi$ and π.

d. Fitness Function

The fitness function is a function that converts the value into a fitness value.

e. Fitness Value

The fitness value is a number that describes how close the chromosome is to the optimal solution. The solution could be a maximum, minimum, local maxima or local minima, depending on the problem being solved. Identifying a local maxima or minima is of interest when utilizing a GA in a cost savings problem.

f. Population

The population is a group of elements that exist within the genetic algorithm.

g. Generation

Genetic algorithms operate iteratively. Each iteration is referred to as a generation. Generally, the population at the start of a generation is the population at the end of the previous generation. For the case of the first generation, the population is randomly generated.

h. Survival of the Fittest

Survival of the fittest is a process by which chromosomes are selected by their fitness value. During this process, some elements are removed from the population. This is analogous to nature in that the weaker species die off.

i. Crossover

Crossover is the process by which two elements that were selected during survival of the fittest combine to produce two new elements. The combination occurs by randomly picking a gene position. All of the genes to the left of this gene in chromosome

a are combined with all of the genes to the right of the same position in chromosome b to create a'. This process repeats to create b' with the unused portions of a and b. This is analogous to nature in that two parents with desirable characteristics might go on to produce children that also have desirable characteristics.

j. Selection

There are several ways to determine which of the strings are selected for crossover. One method is to only take the elements with the best fitness values. Unfortunately this has the effect of removing some chromosomes from the solution pool that might actually be needed to arrive at the optimal solution. This can be countered through the use of the "roulette wheel" algorithm which assures that all elements of the population have at least a small chance of being selected.

k. Mutation

Mutation is the process by which the genes of a single element may be changed by a random process. This is analogous in nature to an event causing a gene to change in an animal thus making it more suited for survival. An example in the following section demonstrates the need to implement mutation.

l. String Generation

In a GA, several different chromosomes exist at one time. This collection is referred to as the population. The implementation of the GA used in this thesis has a population of 16, based on the number of items that can easily be sorted with the Batcher sort. Initially, the population is created through a random string generator. Once this has been accomplished, each of the strings is evaluated with the fitness function. The results of these calculations are then sorted from the highest fitness value to the lowest. This ordering is then used to help determine which of the strings will be selected for crossover. As such, this implements the concept of "survival of the fittest".

2. Example of a Genetic Algorithm

The following discussion on the implementation of genetic algorithms is based on GAs taken from [4]. In order to solve a program using GAs, the potential solution must first be encoded as a chromosome. Consider a problem of finding the maximum value of $f(x) = x^2$ for $0 \le x \le 4,095$. Assume x is realized as a 12-bit binary number. Its value is directly derived by converting the chromosome into an integer. Its fitness function is simply the square of its value. Thus, the higher the fitness function result, the closer to the solution you are. Obviously, the best solution to this problem is the string with 12 ones in it. Figure 9 shows the processes by which the GA operates. Finally, the example will not implement mutation, and while doing so demonstrates its need in order to achieve the maximum.

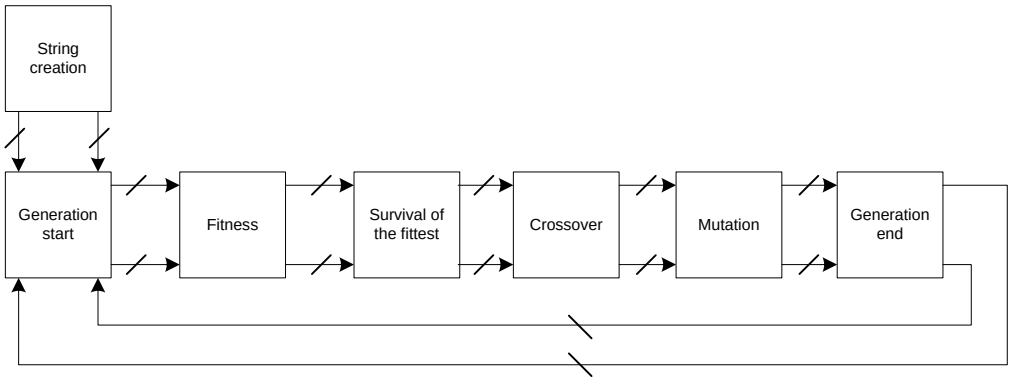

Figure 9 GA algorithm

The GA starts with the creation of random strings, which represent each member in the population. Each of the elements then has their values and fitness values calculated as shown in Table 5.

Member	Chromosome	Value	Fitness value
1	110101100100	3,428	11,751,184
2	010100010111	1,303	1,697,809
3	101111101110	3,054	9,326,916
4	010100001100	1,292	1,669,264
5	011101011101	1,885	3,553,225
6	101101001001	2,889	8,346,321
7	101011011010	2,778	7,717,284
8	010011010101	1,237	1,530,169

Table 5. Start of 1st generation, From [4]

Initially, the population consists of the 8 randomly generated chromosomes. Through survival of the fittest, members 1, 3, 6 and 7 are chosen, since they have the highest fitness value. Members 1 and 3 are then chosen randomly to crossover at the second bit from the left, while 6 and 7 crossover at the 6th bit. These positions are marked in the following table with a "/" as shown in Table 6.

Member	Chromosome	Value	Fitness value
1	11 / 0101100100	3,428	11,751,184
2	10 / 1111101110	3,054	9,326,916
3	101101 / 001001	2,889	8,346,321
4	101011 / 011010	2,778	7,717,284
5	111111101110	4,078	16,630,084
6	100101100100	2,404	5,779,216
7	101101011010	2,906	8,444,836
8	101011001001	2,761	7,623,121

Table 6. Start of 2nd generation, From [4]

By applying only survival of the fittest to the start of the 2nd generation, the following table is constructed. In Table 7, it can be seen that the least significant bit in all of the chromosomes is zero. As previously mentioned, the maximum value for this GA is a string with all ones. Thus, regardless of where crossover is performed, the least significant bit in each chromosome will remain zero preventing the maximum from being achieved. In order to prevent this from happening, mutation is necessary.

Member	Chromosome	Value	Fitness value
1	110101100100	3,428	11,751,184
2	101111101110	3,054	9,326,916
3	111111101110	4,078	16,630,084
4	101101011010	2,906	8,444,836

Table 7. After survival of the fittest, 2nd generation, From [4]

3. Advanced Operations

a. Selection Methods

Previously, when parents were selected for crossover, they were chosen only by their fitness value. This prevents good genes that exist in chromosomes with a poor fitness value from propagating themselves into later generations. By providing a detailed selection method, a means will exist that makes it possible for good genes in bad chromosomes to propagate.

The roulette wheel algorithm is a process by which any of the chromosomes may be selected for crossover. It is based on an idea that the chromosomes are chosen with a probability that depends on their fitness value. Additionally, it allows for chromosomes with poor fitness values to be selected, albeit considerably less frequently.

It begins by determining the sum of all of the fitness values, which are assumed to be non-negative. Next, a random number is generated that is between 0 and the sum of the fitness values. Next, a running total is initialized to 0, and each member of the population has its fitness value added to it. The fitness values of subsequent members of the population are added to the total until the running total is equal to or greater than a randomly generated number. The last added chromosome is then selected for crossover. This process continues until enough chromosomes have been selected to cause the population to be filled.

b. Elitism

The roulette wheel provides an approach that ensures that any of the chromosomes has the opportunity to reproduce and be part of the next generation. The side effect of this is that sometimes an ideal solution is removed from the population. Elitism is the concept that prevents this from happening. It allows certain solutions which meet specified criteria to remain in the population. This process overcomes its not being selected for crossover, or mutation changing a gene [4].

c. Selective Crossover

As previously discussed, one step in crossover is to randomly select a point at which to perform crossover. Consider the traveling salesman problem in which a salesman needs to travel through a series of cities in the shortest possible trip. Suppose that there are six cities that need to be visited, named a, b, c, d, e and f. The chromosome is composed of the order in which the cities are visited. Thus, two possible chromosomes would be *abcdef* and *debcfa*. If these two chromosomes were crossed in between the third and fourth genes, the resulting chromosomes are *abccfa* and *debdef*. In each of these cases, two cities are visited twice during the tour, and neither is a solution to the problem. Because of this issue, additional care should be taken when performing crossover to ensure that the resulting chromosomes are valid solutions [4].

B. IMPLEMENTATION ON THE SRC-6

The specifics of the problem described in this section deal with a GA that solves a packing problem. Constructing a ROTS function, that might be bent, can be viewed as this packing problem. The goal of this packing problem is to find a combination of objects that weigh a total of 28 pounds. There are four types of items to be packed, 9 items weighing 6 pounds, 2 at 3 pounds, 1 at 2 pounds and 2 at 1 pound. This results in a chromosome that can be described with 14 binary digits.

Note that we can divide this problem into two subproblems. The first subproblem is to create a subtotal of 4 pounds. This can come from two possibilities. The first is that exactly one 3 pound object and one 1 pound object are chosen. The other possibility is that the 2 pound and both 1 pound objects are chosen. The other subproblem is to create a subtotal of 24 pounds. There are also two ways to do this—four 6 pound, or three 6 pound and both 3 pound objects. In the latter case, this prohibits the use of the three pound and one pound objects to create 4 pound subtotal. This process describes the fitness function. Each subproblem contributes a score of 120 to the fitness value, thus resulting in an optimal fitness value of 240. All other combinations of selected objects result in a fitness value of less than 240. For example, consider three chromosomes that

contain only 6 pound objects. The first, second and third chromosomes contain three, four and five 6 pound objects respectively. Since the first and the last chromosomes do not have the required number of 6 pound objects, their fitness sub-value would be 90. The second chromosome, which contains the correct number of 6 pound objects, would thus have a fitness sub-value of 120. This process holds true for all combinations of genes, and will not be discussed further.

Figure 10 shows the data flow path for the GA and the elements described in the previous section on GA are implemented. The sorting function is the method used utilized to facilitate survival of the fittest.

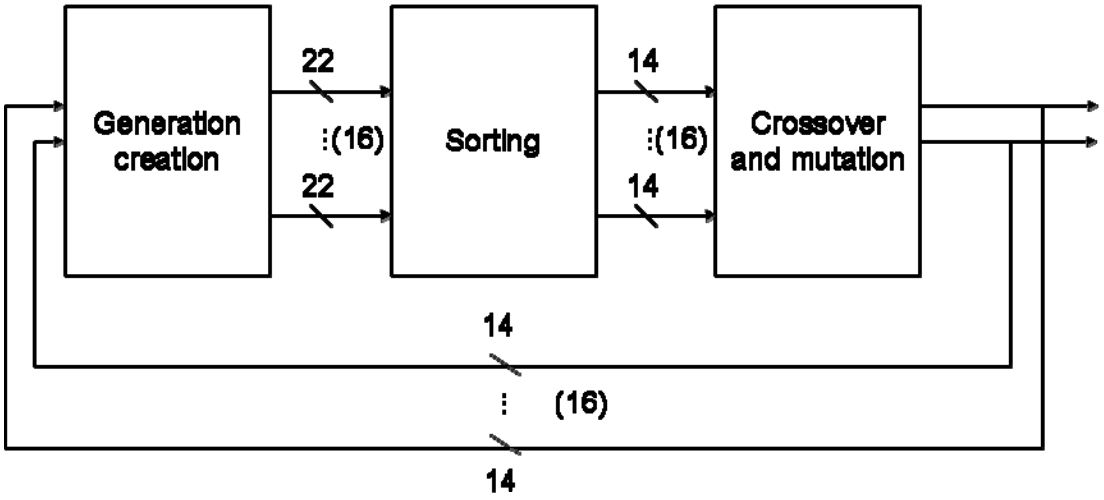

Figure 10 GA implementation organization

1. Generation Creation

Generation creation includes several processes. There are three primary functions. The first is the construction of new chromosomes from pseudo-random numbers. Second is the calculation of the fitness value for each of the chromosomes. Finally, the third process is ensuring genetic diversity.

a. Generation Creation

Generation creating is achieved through the circuit in Figure 11. This circuit is representative of how each of the 16 different chromosomes is created for the

GA. The number of chromosomes is based on the ability to apply the Batcher sort to them. A larger population is possible if the Batcher sorting module is expanded to accommodate the population. Furthermore, it shows how the clear unit introduces new strings into the population. A linear feedback shift register (LFSR) initializes the population with pseudo-random chromosome strings. The LFSRs are initialized through a clear signal generated during the first generation. Additionally, for each position in the population, the corresponding chromosome from the previous generation has its fitness value determined. Once the fitness value is created, it is appended onto its chromosome's associated bit string.

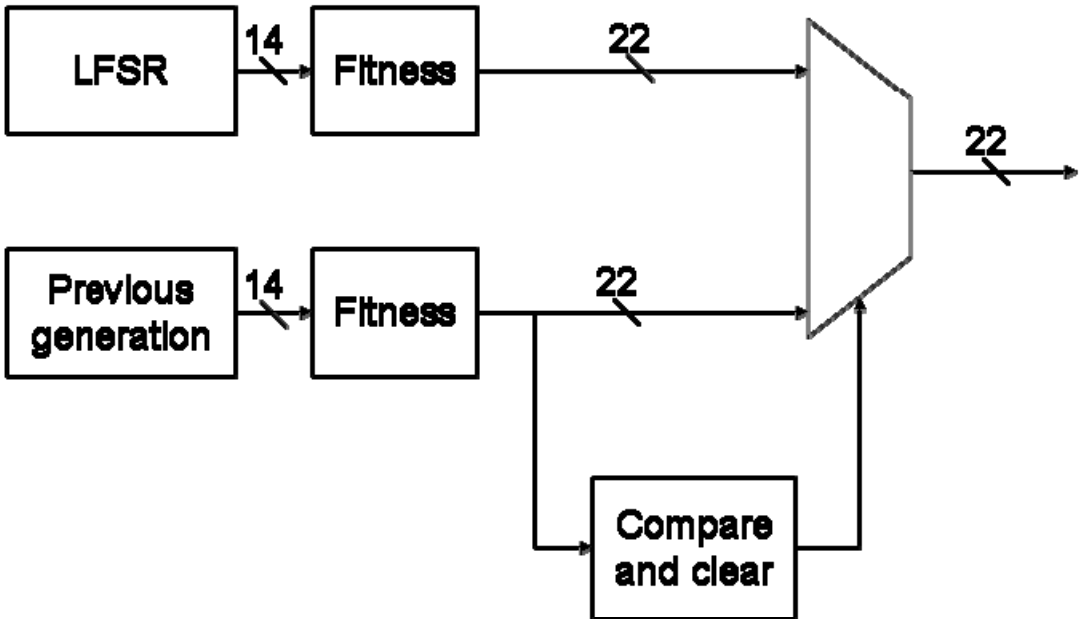

Figure 11 Generation creation

b. Compare and Clear Unit

The compare and clear unit evaluates the previous generation fitness function against a user-specified constant, which is capable of being specified at the keyboard. It serves to replace (clear) the chromosome from the previous generation, if it does not reach a threshold value, with a new chromosome generated by the LFSR. This process helps the implementation of the survival of the fittest concept previously discussed. The CLEAR signal that is generated during the first generation is also used by

this unit causing string initialization when the GA starts. Finally, it is also capable of generating additional clear signals can be generated through the half-life and Order 67 circuits.

c. Half-life

Half-life operates on the idea that, on each generation, the chromosome at a particular element position is loaded into successive registers. If those registers contain the same value over 3 generations, a clear signal is generated forcing a new chromosome into that position in the generation. The "3 generations rule" was arbitrarily chosen to allow sufficient time for fit chromosomes to crossover and propagate throughout the population. This implementation is shown in Figure 12.

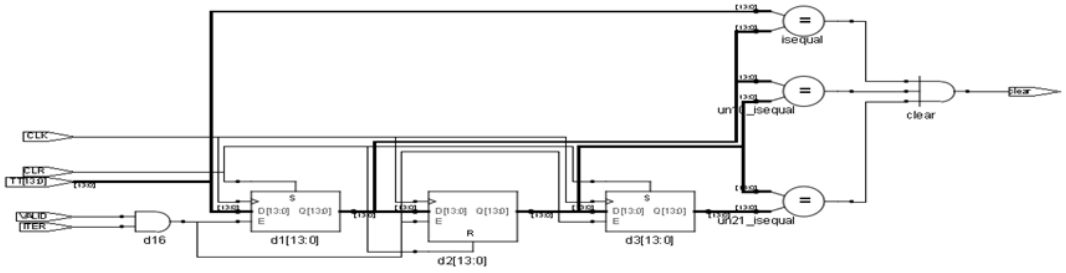

Figure 12 Half-life circuit

d. Order 67

Order 67 is based on the idea that if adjacent members in the top four chromosomes in the population are the same (clones), there is a lack of diversity, and thus, a clear signal should to be generated in order to replace an old chromosome by a new one. This is of concern due to the small population size. This is implemented through a set of simple comparison circuits whose output drives an OR gate that provides the CLEAR signal as shown in Figure 13.

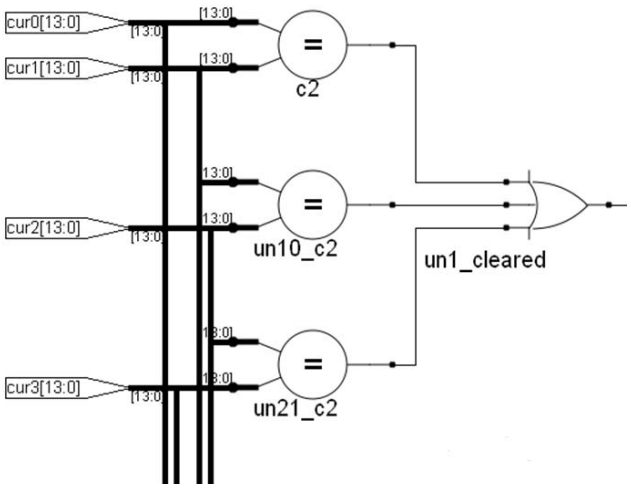

Figure 13 Order 67

e. String Generation

The circuit in Figure 14 is an example of a general purpose LFSR that was used in EC4830 from the course notes, and from an exam question. Its primary advantage is that it can produce a maximal run sequence provided the correct tap positions. The number of different outputs that an LFSR can produce is dependent on its tap positions. Since an LFSR requires at least 1 bit to be one at all times during its operation, the maximal run of an n bit LFSR is $2^n - 1$.

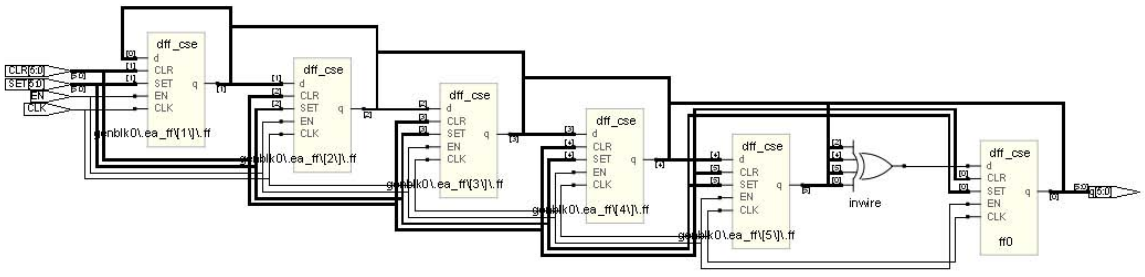

Figure 14 General case LFSR

The code below shows how the LFSR is parameterized. The parameter n determines the number of bits in the shift register. The tap parameters allow for up to 4 taps, dependant on the particular LFSR being implemented. The tap position came from

a predetermined table [17]. The module that instantiates the LFSR module specifies the tap parameters. Therefore, tap parameters are not shown in this section of code. When implemented, the tap positions were at 13, 4, 2 and 0. Figure 15 is the Verilog code that was written to realize the general case LFSR shown in Figure 14.

```
module LFSR(CLR, SET, EN, CLK, q);
	parameter n=6;
	//parameter taps=4;
	//Parameter n corresponds to the number of bits in the LFSR
	//Each of the tapX parameters directly corresponds to the maximal tap
	//as shown in Table 3.8 of Dixon

	//In order to properly use the LFSR, the register must first be initialized with the
	//CLR and SET inputs

	//The value in SET is the first value stored in the register
	//The value of CLR must be the NOT of SET
	//SET and CLR must be LOW after initialization for the LFSR to sequence


	parameter tap0=0;
	parameter tap1=0;
	parameter tap2=0;
	parameter tap3=0;
	input [n-1:0] CLR, SET;
	input EN, CLK;
	output [n-1:0] q;
	wire [n-1:0] q;
	wire inwire;
	assign inwire = q[tap0]^q[tap1]^q[tap2]^q[tap3];
	dff_cse ff0(inwire, CLR[0], SET[0], EN, CLK, q[0]);
	genvar k;
	generate
	for (k=1; k<n; k=k+1)
		begin: ea_ff
			dff_cse ff(q[k-1], CLR[k], SET[k], EN, CLK, q[k]);
		end
	endgenerate
endmodule
```

Figure 15 General case LFSR Verilog code

Figure 16 shows the code that is necessary to instantiate the LFSRs. Since the LFSR needs to retain its state from one cycle to the next, it uses the VALID and ITER control signals to allow the LFSR to only shift state once per call to the macro. The inputs to the macros allow for different random number seeds to be specified from main.c.

```
module lfsrs(rnd0, rnd15, CLR, VALID, ITER, CLK, w_rng0, w_rng15);

    parameter n=6;

    parameter AAA=5;
    parameter BBB=4;
    parameter CCC=2;
    parameter DDD=0;

    input [n-1:0] rnd0, rnd15;
    input CLR, VALID, ITER, CLK;

    output [n-1:0] w_rng0, w_rng15;
    wire [n-1:0] w_rng0, w_rng15;

    wire [n-1:0] clears [15:0];
    wire [n-1:0] sets [15:0];


    assign sets[0]=rnd0;
    assign sets[15]=rnd15;
    assign clears[0]=~sets[0];
    assign clears[15]=~sets[15];

    defparam rng0.n=6;
    defparam rng0.tap0=AAA;
    defparam rng0.tap1=BBB;
    defparam rng0.tap2=CCC;
    defparam rng0.tap3=DDD;
    LFSR rng0(clears[0]&{6{CLR}}, sets[0]&{6{CLR}}, VALID&ITER, CLK, w_rng0[5:0]);
    defparam rng15.n=6;
    defparam rng15.tap0=AAA;
    defparam rng15.tap1=BBB;
    defparam rng15.tap2=CCC;
    defparam rng15.tap3=DDD;
    LFSR rng15(clears[15]&{6{CLR}}, sets[15]&{6{CLR}}, VALID&ITER, CLK, w_rng15[5:0]);

endmodule
```

Figure 16 LFSR instantiation code

f. Numerical Representation

As previously mentioned, there are 14 genes in each chromosome for this GA. The genes are organized in the chromosome such that genes representing an object with the same weight are adjacent to one another. Table 8 shows an example of the layout of the chromosome's genes with respect to its weight.

Bit	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Weight	6	6	6	6	6	6	6	6	6	3	3	2	1	1

Table 8. Chromosome format

g. Fitness Function

The fitness function was described briefly in the introduction to this section. The flow chart in Figure 17 is representative of its implementation in Verilog. The figure refers to lookup tables that will not be discussed in detail. The basis for the tables is on how much of each of the fitness sub-problems has been correctly solved. The maximum fitness is 240 and the minimum is 0. The maximum is based on each of the sub-problems receiving a fitness sub-value of 120. The width of the fitness value is 8 bits.

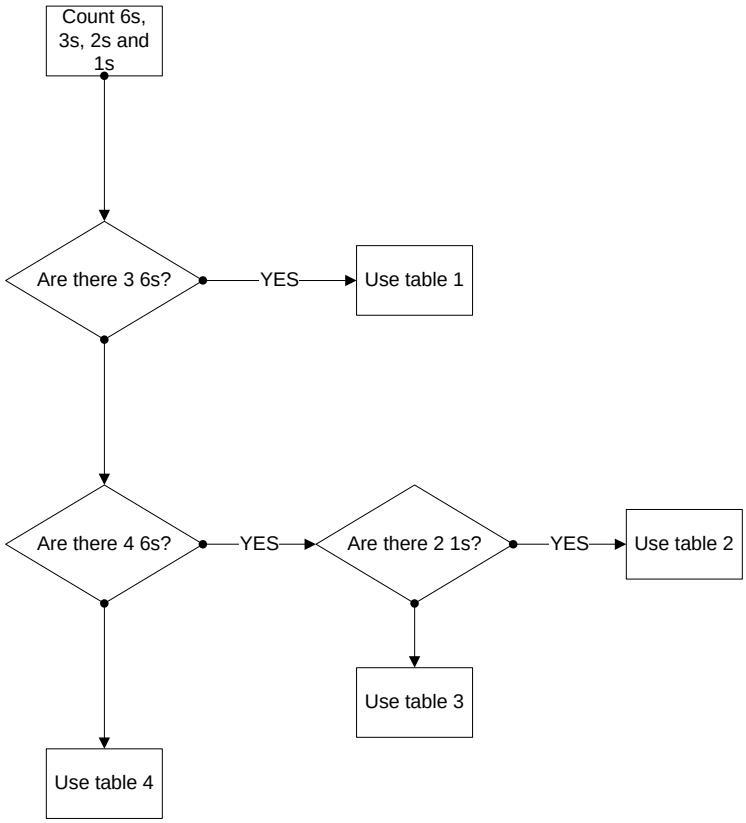

Figure 17 Fitness function flowchart

2. Sorting

The need to sort the fitness values in the population is complicated by two factors. When the fitness values are sorted, their corresponding chromosomes must also be swapped. If this does not happen, the fitness values would lose their meaning. The

second factor is based on the nature of operating the GA on an FPGA – that is the need to sort the chromosomes in hardware. This is why a parallel sort, such as the Batcher sort, was chosen [1]. Furthermore, because it is implemented on specialized hardware, an FPGA, it is able to take full advantage of the parallelism of the Batcher sort.

Figure 18 is taken from [9]. Each horizontal line represents an element in the population. The vertical arrows represent a comparison and swap, when required, between the two elements. The well-trained eye will notice the symmetry involved in this sort. Simply put, sorting 16 elements first requires the sorting of two sets of 8 elements and then a merging of the two sets. Likewise, sorting 8 elements first requires sorting two sets of 4 elements, and so forth. Of particular concern is ensuring that all paths through the sorting network have the same pipeline length. This is easily seen on element e_0. After the two 8 element sets are sorted either e_0 or e_8 is the largest. Once they are compared, and swapped if required, no further comparisons need to be made with e_0. Since, after each comparison and swap is made, the resultant values are loaded into registers, those elements not compared in a clock cycle must also be loaded into registers to ensure all data pipelines are of equal length. There are 4 comparisons in the shortest path, and 10 comparisons in the longest path.

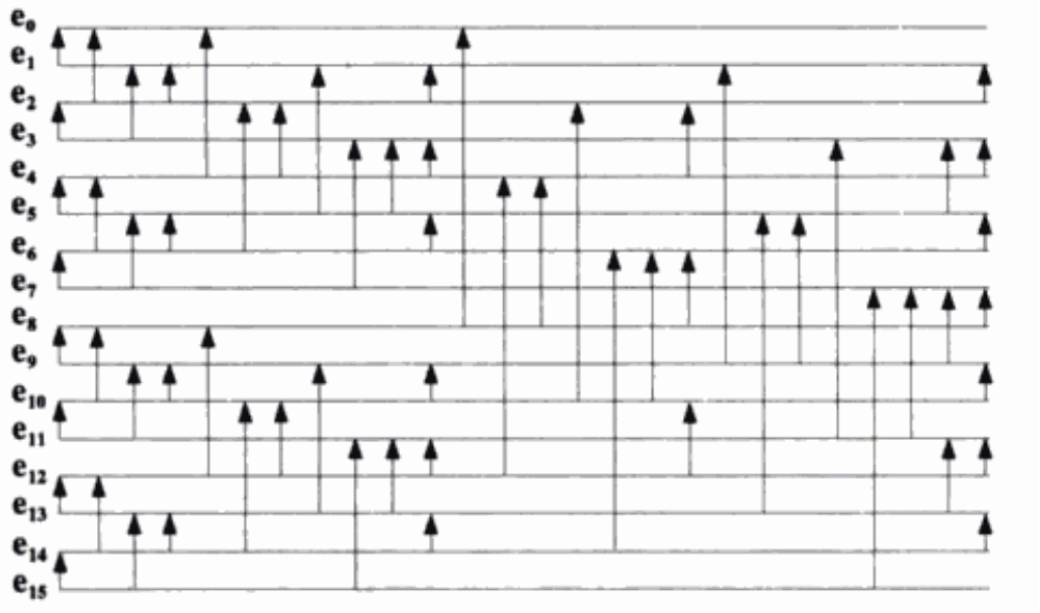

Figure 18 16 element Batcher sort, From [9]

Each arrow in Figure 18 is realized by the sorting element shown in Figure 19. As previously mentioned, it is important to maintain the relationship between the chromosome and its fitness value when the fitness value is sorted. The comparison between the bits representing the fitness values determines whether or not a swap will be made.

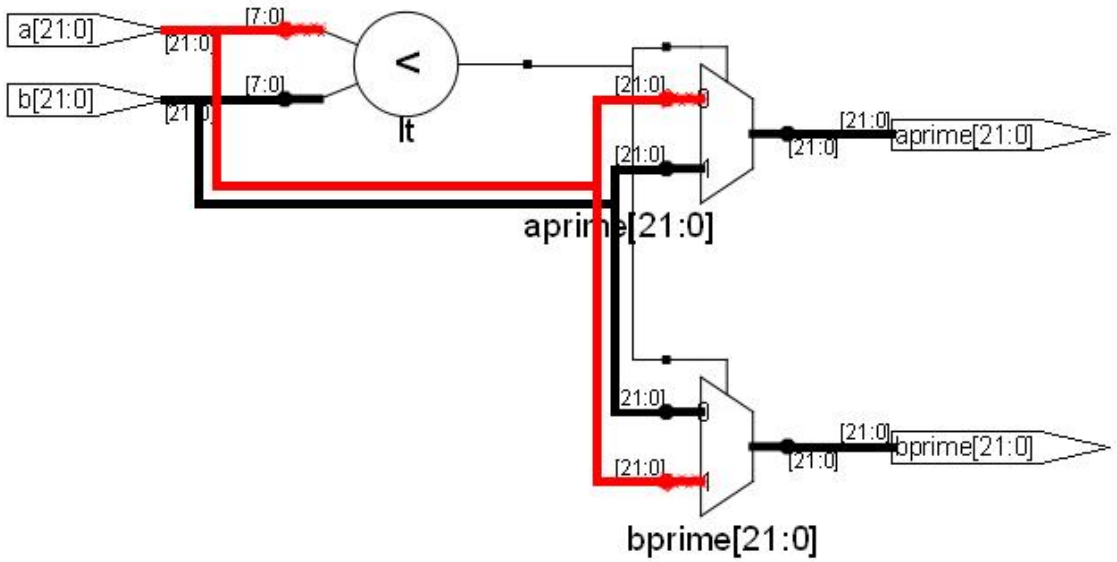

Figure 19 Swapping element for sorting

3. Crossover and Mutation

There are two main parts to the crossover and mutation section, namely crossover and mutation. There is an additional helper module that provides a scalable means to increment the addresses of a ROM.

a. ROM Address Control

Due to the use of several ROMs in this GA, a common circuit was created to control accessing their elements and is shown in Figure 20. This allows easily scaling the GA to allow ROMs containing more words. The first element is an adder, which merely adds 1 to the previously used address. The multiplexer is controlled by the CLEAR signal that is generated during the first generation. When it is high, the output of

the multiplexer is 0, thus providing a means to initialize the address register. As previously mentioned, the address register provides the adder with the address to be incremented, and which value from the ROM that is to be accessed.

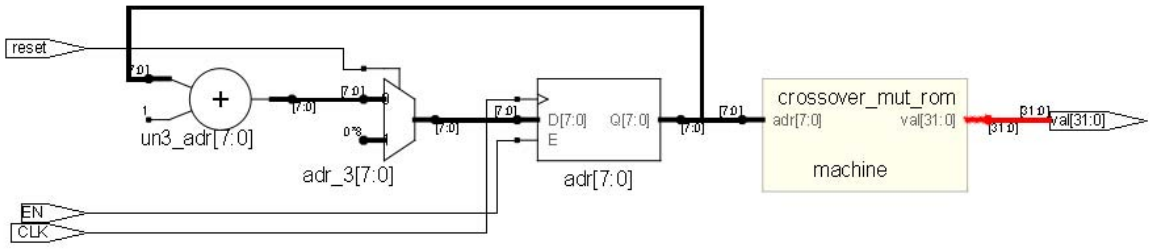

Figure 20 ROM address control

b. Crossover

As previously mentioned, the need to select members of the population for crossover is essential for proper operation of the GA. Although the roulette wheel method provides for a good way to select parents, it is difficult to implement in the constant time needed for this pipelined problem. Furthermore, the need to generate random numbers across a range that is unspecified until all strings have been created is difficult, if not impossible, to accomplish in Verilog. In order to combat this, a ROM was created that picked which elements would crossover. This was achieved by creating a C++ program that generated 32-bit words. These words consist of 8 nibbles, with each nibble representing which one of the 16 elements in the population is selected for crossover. The selection probability can be changed to whatever is desired for experimentation.

The roulette wheel is based on the desire that each element of the population has a chance at being selected for crossover. The sum of all of the fitness values is first determined. Next a random number is generated between the 0 and this sum. A running total is next initialized to zero. The fitness value of each of the elements

is added to the running total until the running total meets or exceeds the random number generated. When this happens, the last element added to the running total is selected for crossover.

The criterion for the formula that was implemented was to provide a rough approximation of the roulette wheel method, albeit not at the same proportions as described by the roulette wheel method. At the output of the sorting circuit, the elements are sorted from most fit to least fit. They are then assigned a name, with 0 indicating the best fitness value, and 15 the worst. A list was created from which each element is selected. The composition of the list is initially 16 0's, 15 1's, …, and 1 15. This distribution allows all elements to be selected for crossover, while favoring those elements with the best fitness values. Initially, the probability that the best fitness value is selected for crossover is 11.7%. Likewise the second best and worst fitness values have a probability of 11.0% and 0.7%, respectively. The elements of the list are then shuffled and the first element is selected as the first nibble in the word that was being stored in the ROM. The list is then parsed removing all copies of the element that was just selected. Thus, if the fittest element is selected on the first choice, the second fittest element now has a probability of 12.5% of being selected, while the worst fit element has a probability of 0.8%. Generally, GA implementations allow an element to be selected for crossover multiple times. However, due to the small population size, this implementation removes elements selected for crossover to help force genetic diversity. This process continues until eight elements are selected. The remainder of the C++ program creates the structure that wraps the ROM values with the necessary Verilog code to facilitate its instantiation. An example of one word that would be stored in this ROM is 32'b0000_0001_0011_0010_1000_0110_0100_1110. When this ROM is read, elements 0, 1, 3, 2, 8, 6, 4 and 14 would be selected for crossover.

All crossover operations are contained within the crossover module. It consists of a crossover ROM and 4 crossover units. The ROM provides a control signal to the two multiplexers in the crossover units directing which elements will be selected for crossover. The crossover units process the selected elements to create the children. A crossover module is shown in Figure 21.

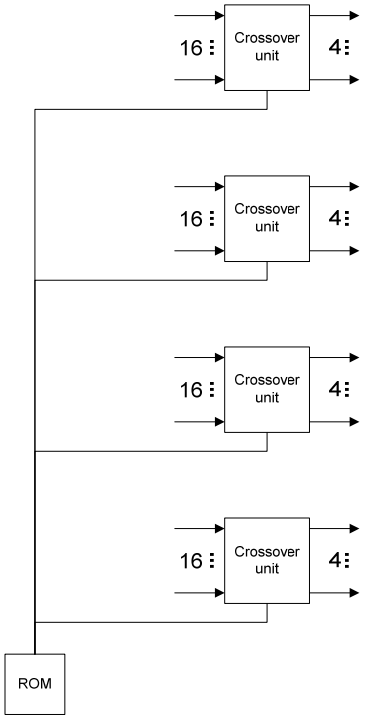

Figure 21 Crossover module

Since crossover and mutation destroy the fitness value of the chromosomes, the bits representing the fitness value are discarded. Each crossover unit consists of 2 16-to-1 multiplexers and a bit swapper module. Each of the multiplexers is controlled by one of the nibbles produced by the previously mentioned ROM. The output of these multiplexers is then applied to the bit swapping module as shown in Figure 22.

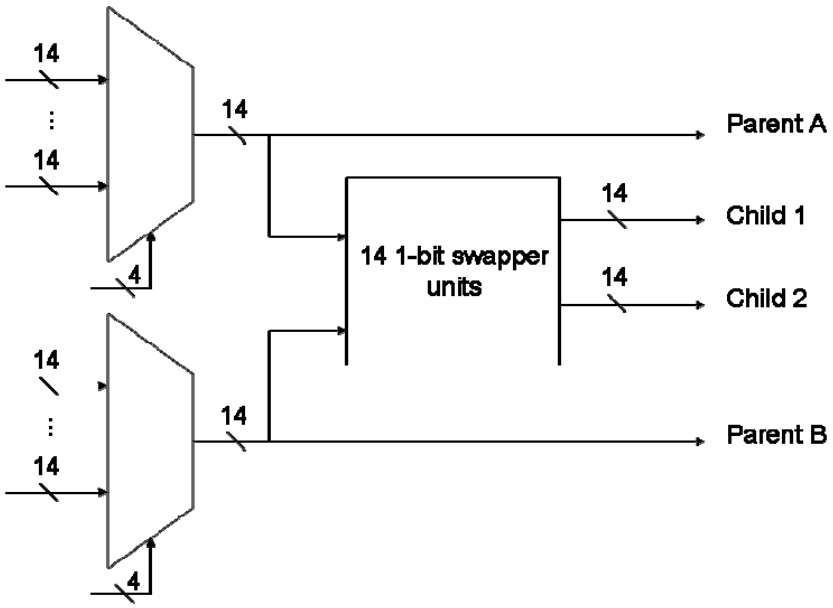

Figure 22 Crossover unit

The bit swapping module consists of fourteen 2 input multiplexers. Each multiplexer receives the same control signal, the crossover code; however, the inputs are reversed in one of multiplexers as shown in Figure 23. This creates a 14 bit swapping circuit where each bit is swapped based on its associated value in the crossover code (the control signal for the multiplexers that perform crossover). This is the same basic idea that is used in the swapper elements for the sorting circuit.

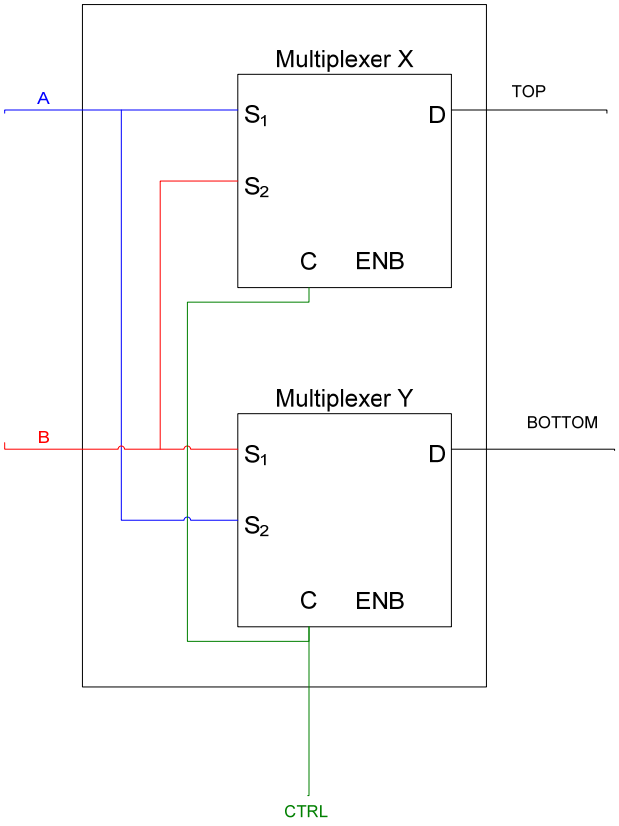

Figure 23 1 bit crossover circuit

As previously mentioned, care should be taken when choosing the bit position where crossover will occur. Since the fitness value is composed of two subvalues, the decision was made to perform crossover at the boundary between the objects with a weight of 6 pounds and those of 3 pounds. Consider two elements with relatively high fitness values. One could meet the criteria of having four 6 pound objects selected, and the other chromosome meets the criteria of having a combination of objects selected that total 4 pounds. Then, these two elements crossover at the aforementioned boundary, one of their children would have the optimal solution. In order to provide greater flexibility, and to demonstrate the effects of choosing a crossover point, the crossover code is capable of being specified by the user at run time.

c. Mutation

Although it was previously shown that mutation is essential to GAs, it is nonetheless a rare occurrence. Since mutation is essentially the inversion of a bit, this is most easily accomplished through an exclusive OR gate. In this implementation of the GA, the frequency of mutation is composed of two factors, how often a mutation might take place, and how likely it is that each individual bit will mutate. The probability of mutation is usually on the order of 1 bit out of a 1,000; however, like all aspects of the GA, each of the parameters is very dependent on the problem [4]. Since this research does not expect mutation to play a large role due to the design of the chromosome and fitness function, mutation is implemented to demonstrate proof of concept. For example, suppose that there is a 1% chance that a mutation will take place during a generation. For each time that a mutation takes place, each bit has a certain probability, for example 10%, that it will mutate. This means that a mutation might be "scheduled" to take place in which no mutation actually occurs.

A mutation string could be shown to be the 14-bit string that represents which bits will be inverted. Because of the difficulties that are involved in making these determinations in Verilog, these mutation strings are most easily stored in a ROM. For each generation in which mutation does not take place, the 14-bit string is 0. This implies that one could have $14x100 = 1,400$ consecutive zeroes before the occurrence of a one. This means that the mutation ROM is a good candidate for compression.

The compression for mutation was decided to be as simple as possible to provide at least the proof of concept. It was then decided to create the mutation string as follows: 7-bits of zero run length, and 14-bits of mutation code and 4-bits of element selection. The seven bit length is based on a given probability that there is a 1% chance that mutation will occur during a generation. This means that, over 128 generations, it is likely that a mutation will occur. However, since this is not guaranteed, the exception to this case will be discussed later.

Again, the ROM is generated as Verilog code using a C++ program. Random numbers are generated with the specification that 1% of the strings will mutate.

For each bit, there is a 10% chance that it will mutate, which is represented by encoding a one. In the cases in which 128 generations pass without a mutation, the mutation code is automatically created as a 14-bit 0 vector. Finally, the 4 bits to select which element of the population is to be mutated are randomly created in the C++ program.

Figure 24 shows the operation of the mutation circuit. It consists of four main parts. Starting from the right, a multiplexer that is controlled by how many generations have passed since the last mutation. If the GA is ready to mutate, then its output is the mutation code and element selection from the ROM; otherwise it is zero. Next is the mutation ROM. The second element is an incrementer that controls the address lines for the ROM. The first element is another incrementer that counts the number of generations since the last mutation. It compares that number with the zero run length value stored in the ROM, and generates the control signal for the multiplexer controlling the mutation code. The remaining circuitry is provided to control initialization and operation of the circuit.

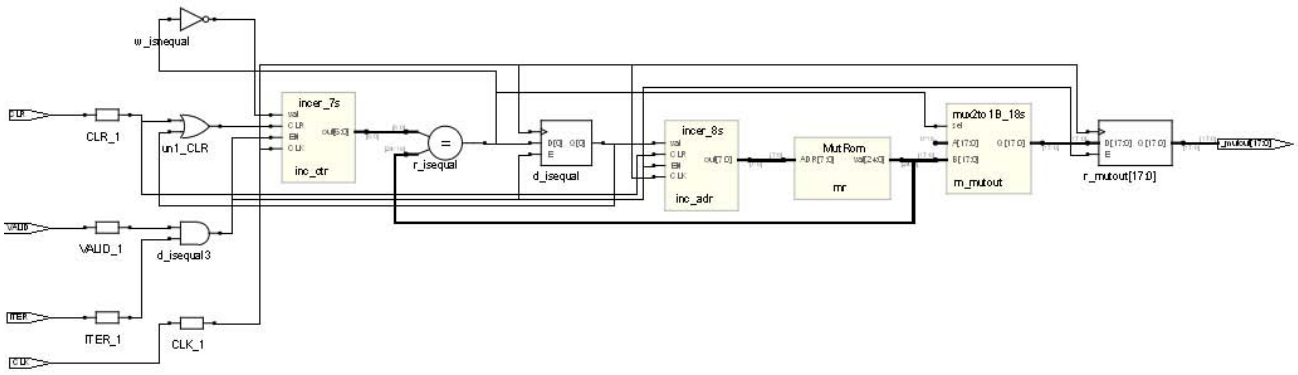

Figure 24 Mutation ROM

C. ADVANCED IMPLEMENTATION ISSUES

1. Circuit Reutilization

One of the key elements to the GA is the fitness function. The current implementation of the GA includes 16 elements. Each of these elements requires two

instances of the fitness function. The first performs the calculation of the fitness value from the previous generation, and the fitness for the value from the LFSR. This is made possible due to the small size of the fitness function. Unfortunately, if the fitness function was chosen as the nonlinearity calculation, thus allowing searching for bent functions over all of the search space of $n = 6$, the available resources on the FPGA would be quickly exhausted. Table 9 shows the utilizations for two instances and eight instances of the $n = 6$ nonlinearity calculation. Of further clarification, no other circuitry is involved with these utilizations.

	2 calculations	8 calculations
Number of Slice Flip Flops	12%	33%
Number of 4 input LUTs	19%	70%
Number of occupied Slice	28%	94%
Freq	100.1 MHz	100.0 MHz

Table 9. FPGA utilization

Because of the excessive utilization of the FPGA resources, it is necessary to efficiently use the available resources. There are two methods for this. The first is to spread the circuitry over several FPGAs, or through multiple executions, reprogramming the FPGA during execution. For example, moving 128 bits between the macro and subr.mc requires using two 64-bit variables. Unfortunately, the current version of the FPGA C compiler does not provide an efficient mechanism to move large amounts of data between the Verilog module and the C code on the FPGA. This is problematic, since for each 64 bits, or fraction of, one variable must be used to pass the data between the module and the C code, and this must be done for all of the 16 elements. Thus, in order to only determine the nonlinearity of one function on $n = 10$, 16 variables are required to be passed to the macro.

This implies that it might be better to reuse the circuitry that is already laid down on the FPGA. What this means, in the case of the fitness function, is to provide a state machine that will load a specified register on each clock cycle. Figure 25 demonstrates

how this can be accomplished. In this case, the fitness function is substituted by a shift register to show how the different registers can be loaded with an expected value on each clock cycle.

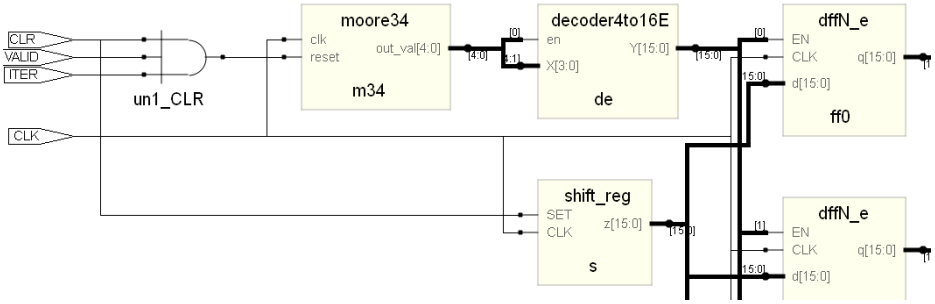

Figure 25 Circuit reutilization

This circuit operates by three main processes. The moore34 module is a state machine. Its output controls the 4 to 16 decoder with enable circuit. The decoder circuit operates as a one hot decoder, if its enable bit is set, otherwise, all of its outputs are 0. Its outputs are used to enable one of 16 registers that are loaded by a common function, in this case a shift register. The shift register is initialized on the CLR signal from a stateful macro. A discussion of stateful macros is included in Appendix A. The state machine that was previously mentioned is described in Figure 26.

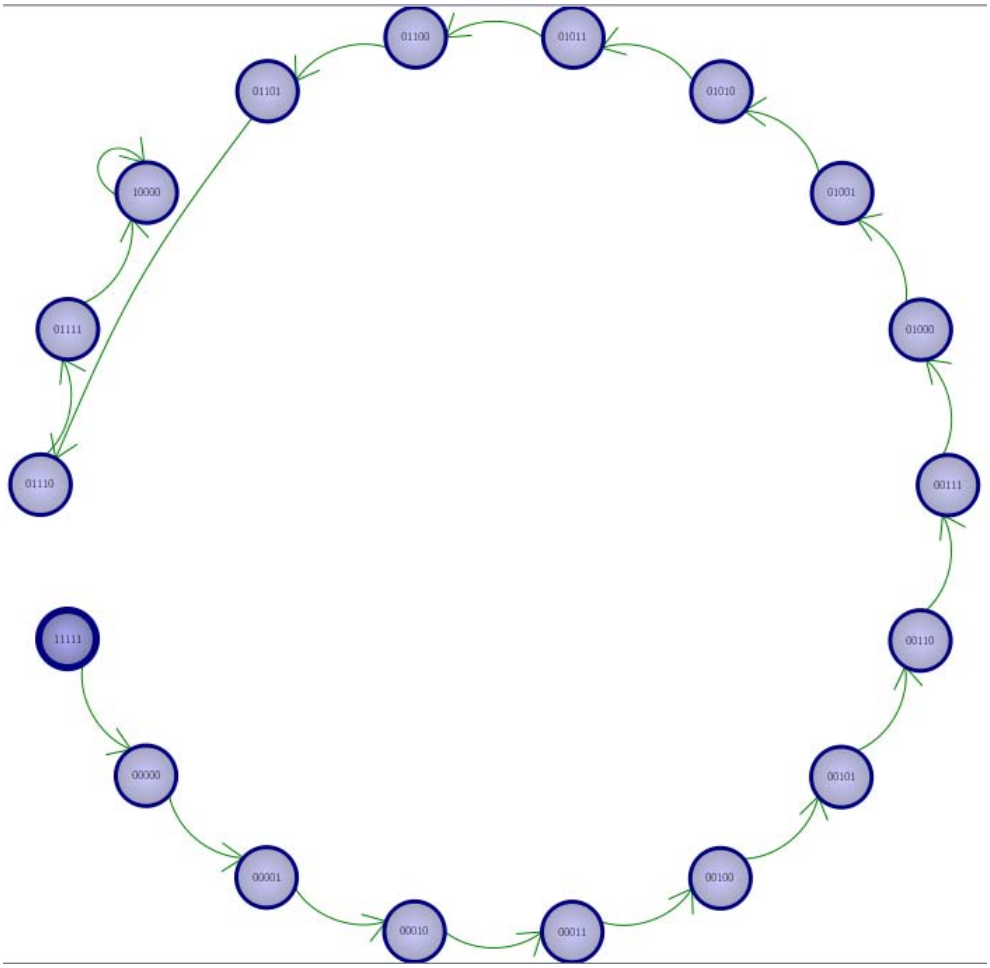

Figure 26 Reutilization state machine

The basic premise of the state machine can be determined by examining the bit code for each state as shown by the number inside each state. The MSB is inverted and applied to the enable line on the decoder. The remaining four bits are then directly applied to the decoder to determine which register needs to be loaded. When the state machine receives its clear signal, it aligns itself such that the next transition will cause register 0000_2 to be loaded. Each successive register is then loaded until all have been loaded. The machine then loops until the next CLEAR signal is received causing the machine to reset. When this machine is implemented with an actual fitness function care must be taken regarding the latency of the fitness function. Its latency must match how long the machine cycles in an initial delaying state until a fitness value is propagated through its pipeline and is ready to be loaded onto an element register.

2. Random Access to ROMs

In order to provide greater variation in program execution, the aforementioned ROMs should not be accessed sequentially, but instead as randomly as possible. The method that was tested to implement this is through a 32-bit cyclic-redundancy check (CRC).

To provide another element of randomness, the value that was calculated via its CRC is calculated by a random number from main.c and added to a timer value from subr.mc. Because accessing the timer prevents pipelining within a loop, the aforementioned sum is incremented during each loop iteration. This provides a randomization of accessing the ROM.

Because of the fixed width nature of the data whose CRC is being calculated, a table lookup CRC was implemented. This allows calculation of the CRC one byte at a time, thus reducing by a factor of four the computation time when compared to a CRC calculation done on a bit-by-bit basis [11]. The CRC lookup table was generated by C code using the method described by [20]. CRC calculation was performed by unrolling a CRC calculation loop of 32-bits and translating the resultant C code into Verilog. This circuit is realized in Figure 27.

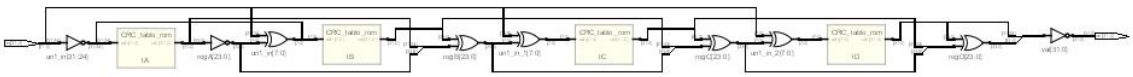

Figure 27 CRC circuit

D. SUMMARY

This chapter describes GAs, with a focus on how they work. The implementation of a GA on the SRC-6 is also given. The weight problem discussed in the chapter is fundamental to the GA that is implemented in this thesis, and will be referred to in subsequent chapters. The next chapter discusses the discovery of bent functions.

IV. BENT FUNCTION DISCOVERY

A. OBSERVATIONS

1. Co-function Repetition

An analysis of the bent functions on $n = 4$ suggests that co-functions might be a means to construct some bent functions. When the 896 bent function on $n = 4$ are analyzed by co-function, it is revealed that there are 112 unique co-functions. By definition, there are two positions where a co-function can exist, the high and the low position. Each of the co-functions occurs 8 times in the high and 8 times in the low positions. This same analysis was conducted on $n = 6$ with similar results. In this case, among the 5,425,430,528 bent functions on 6 variables there are 14,054,656 unique co-functions. Each of the co-functions occur the same number of times in the high and low position. We propose the following:

Conjecture: *All co-functions of n-variable bent functions occur the same number of times in the high and low position among n-variable bent functions.*

2. Index Runs

It has been shown that there are 42,386,176 bent function A-classes on $n = 6$ [12]. We have verified this on the SRC-6. All of these bent functions were enumerated using the degree method described in section II.C.4. Since there are 2^{6+1} bent functions in each A-class for $n = 6$, there are a total of 5,425,430,528 6-variable bent functions. During this, an interesting series of observations were made. First, it is important to understand how the mapping is accomplished. The initial mapping formula is derived from the highest degree of a bent function is $\frac{n}{2}$ and is shown in Tables 10 and 11 [13].

Degree	6	5	5	4	5	4	4	3	5	4	4	3	4	3	3	2	5	4	4	3	4	3	3	2	4	3	3	2	3	2	2	1
ANF bit	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Index	x	x	x	x	x	x	x	15	x	x	x	16	x	17	18	0	x	x	x	19	x	20	21	1	x	22	23	2	24	3	4	x

Table 10. Degree mapping, $n = 6$, Map A, part 1

Degree	5	4	4	3	4	3	3	2	4	3	3	2	3	2	2	1	4	3	3	2	3	2	2	1	3	2	2	1	2	1	1	0
ANF bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Index	x	x	x	25	x	26	27	5	x	28	29	6	30	7	8	x	x	31	32	9	33	10	11	x	34	12	13	x	14	x	x	x

Table 11. Degree mapping, $n = 6$, Map A, part2

The first row in the tables corresponds to the degree of the function based on a bit being present in that position. The second row indicates which bit is being referred to. The bottom row is an index to distinguish between second and third degree functions. For ease of reading, the second degree bits are in red, and the third degree bits are in blue. Those bits corresponding to degrees other than 2 and 3 are indicated by a black 'x'.

Looking solely at the bits corresponding for the second degree functions, we notice the first bit is at location 48 and the second at 40. Their corresponding indices are 0 and 1. This pattern continues until the last 2nd degree function, located at bit 3, is assigned the index of 14. Similarly, the 3rd degree functions begin by again starting from the MSB and working right. Thus, bit 56 has index 15, and 52 has index 16.

The following short hand is introduced to describe the mapping. For example, in Tables 10 and 11, the 2nd degree functions are mapped as 3→14, 5→13, as shown by red circles. Likewise, the 3rd degree functions are mapped as 7→34, 11→33, as shown by blue circles. The map shown above shall be referred to as Map A. Different patterns have been noticed by changing the mapping order. For example, using the above short hand, a new map can be defined as 2nd: 3→34, 5→33 and 3rd: 7→19, 11→18. This mapping shall be known as Map B. Map C is described as 2nd: 3→0, 5→1 and 3rd: 7→15, 11→16. Map D is described, as 2nd: 3→20, 5→21 and 3rd: 7→0, 11→1. Finally a random map was used. These mappings are shown in Tables 12 through 19.

Degree	6	5	5	4	5	4	4	3	5	4	4	3	4	3	3	2	5	4	4	3	4	3	3	2	4	3	3	2	3	2	2	1
ANF bit	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Index	x	x	x	x	x	x	x	0	x	x	x	1	x	2	3	20	x	x	x	4	x	5	6	21	x	7	8	22	9	23	24	x

Table 12. Degree mapping, $n = 6$, Map B, part 1

Degree	5	4	4	3	4	3	3	2	4	3	3	2	3	2	2	1	4	3	3	2	3	2	2	1	3	2	2	1	2	1	1	0
ANF bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Index	x	x	x	10	x	11	12	25	x	13	14	26	15	27	28	x	x	16	17	29	18	30	31	x	19	32	33	x	34	x	x	x

Table 13. Degree mapping, $n = 6$, Map B, part 2

Degree	6	5	5	4	5	4	4	3	5	4	4	3	4	3	3	2	5	4	4	3	4	3	3	2	4	3	3	2	3	2	2	1
ANF bit	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Index	x	x	x	x	x	x	x	34	x	x	x	33	x	32	31	14	x	x	x	30	x	29	28	13	x	27	26	12	25	11	10	x

Table 14. Degree mapping, $n = 6$, Map C, part 1

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

Table 15. Degree mapping, $n = 6$, Map C, part 2

Degree	6	5	5	4	5	4	4	3	5	4	4	3	4	3	3	2	5	4	4	3	4	3	3	2	4	3	3	2	3	2	2	1
ANF bit	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Index	x	x	x	x	x	x	x	19	x	x	x	18	x	17	16	34	x	x	x	15	x	14	13	33	x	12	11	32	10	31	30	x

Table 16. Degree mapping, $n = 6$, Map D, part 1

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

Table 17. Degree mapping, $n = 6$, Map D, part 2

Degree	6	5	5	4	5	4	4	3	5	4	4	3	4	3	3	2	5	4	4	3	4	3	3	2	4	3	3	2	3	2	2	1
ANF bit	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Index	x	x	x	x	x	x	x	5	x	x	x	30	x	22	3	23	x	x	x	21	x	8	1	29	x	7	15	2	6	24	9	x

Table 18. Degree mapping, $n = 6$, Random Map, part 1

Degree	5	4	4	3	4	3	3	2	4	3	3	2	3	2	2	1	4	3	3	2	3	2	2	1	3	2	2	1	2	1	1	0
ANF bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Index	x	x	x	10	x	11	13	26	x	19	27	0	20	16	17	x	x	14	28	32	18	25	12	x	4	33	34	x	31	x	x	x

Table 19. Degree mapping, $n = 6$, Random Map, part 2

This mapping can be extended to higher values of n by using the following technique. To construct the mapping for $n = 7$, the above table is duplicated and appended to the LSB of the original table. The degree value for the original table is then incremented by one and the ANF bits are renumbered in the same manner as the original table, while the duplicated partition is left unchanged. The index of a bit must be

renumbered to achieve unique values. Again, since this thesis only focuses on bent functions of even n, the case of $n = 7$ is only presented as an intermediate step to achieve the mapping for $n = 8$.

Another method that can be used to construct the table is to recognize that the values in the degree row correspond to the number of ones in the binary form of the ANF bit. For example, $63_{10} = 111111_2$ has 6 ones in it, and corresponds to a 6^{th} degree function. Likewise, $42_{10} = 0101010_2$ has 3 ones in its binary representation corresponding to a 3^{rd} degree function.

As previously discussed, this method of mapping the functions by degree allows a search to be conducted only on those functions that can be bent. However, as shown in the above table, this still produces a large search space, 2^{35}. We know that there are 42,386,176 ROTS bent functions on $n = 6$ [12]. The SRC-6 has 6 On Board Memory (OBM) banks located on the MAP, each of which can pass 523,776 64-bit values between main.c and subr.mc. There is not enough OBM to allow storing all of the indices prior to returning them to the microprocessor. Thus, it will take main.c 14 calls to subr.mc in order to retrieve all of the indices representing bent functions. The issue of transferring the indices is further complicated when considering the additional code required to distribute the indices across the six OBMs. Although striping the data across multiple OBMs will make the code more efficient, a less complicated process was chosen to process all of the indices. To overcome this limitation, the search space of 2^{35} is broken into 512 equal parts, called partitions. This number of partitions was chosen to allow a large amount of the loop to be performed on the FPGA, while still having a reasonable assurance the partitions produced a small enough set of bent function indices that they would not exceed the capacity of the OBM. This is done by creating a loop in main.c which called subr.mc. The loop is over the range of 0 to 511. Subr.mc has its own loop that makes $2^{35} / 2^9 = 2^{26}$ calls to a macro. This macro acquires 9 bits from main.c (passed through subr.mc) and 26 bits from subr.mc to form a 35 bit number that represents the ANF of a 3^{rd} or 2^{nd} degree 6-variable function, whose nonlinearity is computed in the macro.

Upon initial execution of the program, and the subsequent data analysis, it was noticed that bent functions commonly occurred as consecutive indices of the 35 bit ANF. For example, suppose the index 656 is discovered to be bent. This is to say, that when the number 656 is applied to the mapping tables above, the resultant ANF is a bent function. The next index, 657, was also discovered to be bent. To completely examine this, a program was written to analyze the indices of bent functions for those cases where consecutive indices yield bent functions. These observations introduce two new terms, index adjacent (IA) bent functions and an IA group. Two bent functions are IA if their index differs by one. A collection of consecutive IA bent functions form an IA group. The length of an IA group is the number of IA bent functions in that group. A future GA may take advantage of this by searching for bent functions by degrees. In doing so, mutation could be implemented by adding one to the index. By doing so, on a bent function, it is possible that the resulting function will also be bent.

Figures 28 and 29 and Table 20 are produced through the analysis of Map A. From these figures and table, we can make some interesting observations. First, note that there are no IA bent functions in partitions above 31. Figure 29 shows how the IA groups are distributed among the 512 partitions. For example, Figure 28 shows that 123,045 bent functions occur in Partition #0, the first partition. Finally, Table 20 shows a table of the length of the IA groups. Second, which only holds true for this experiment, is that all runs exist in the pattern of $2^n - 1, 1 \le n \le 4$. Figures 30 through 35 and Tables 21 through 23 show the corresponding representations for Maps B, C and D.

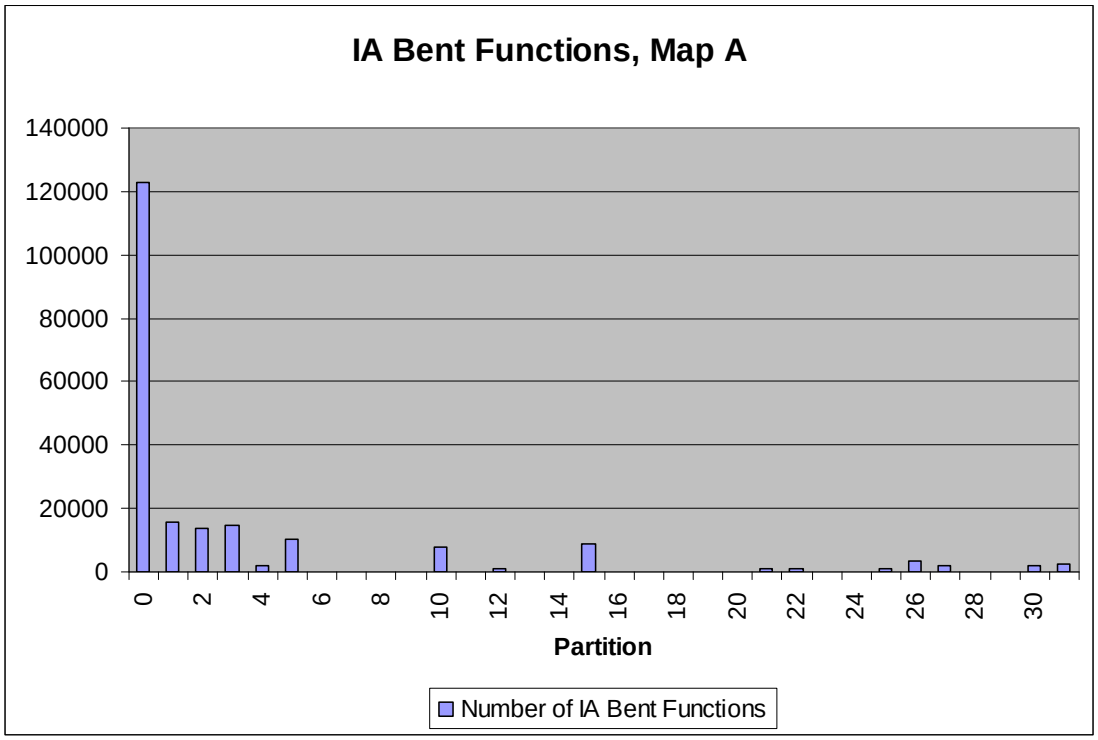

Figure 28 IA bent functions, Map A

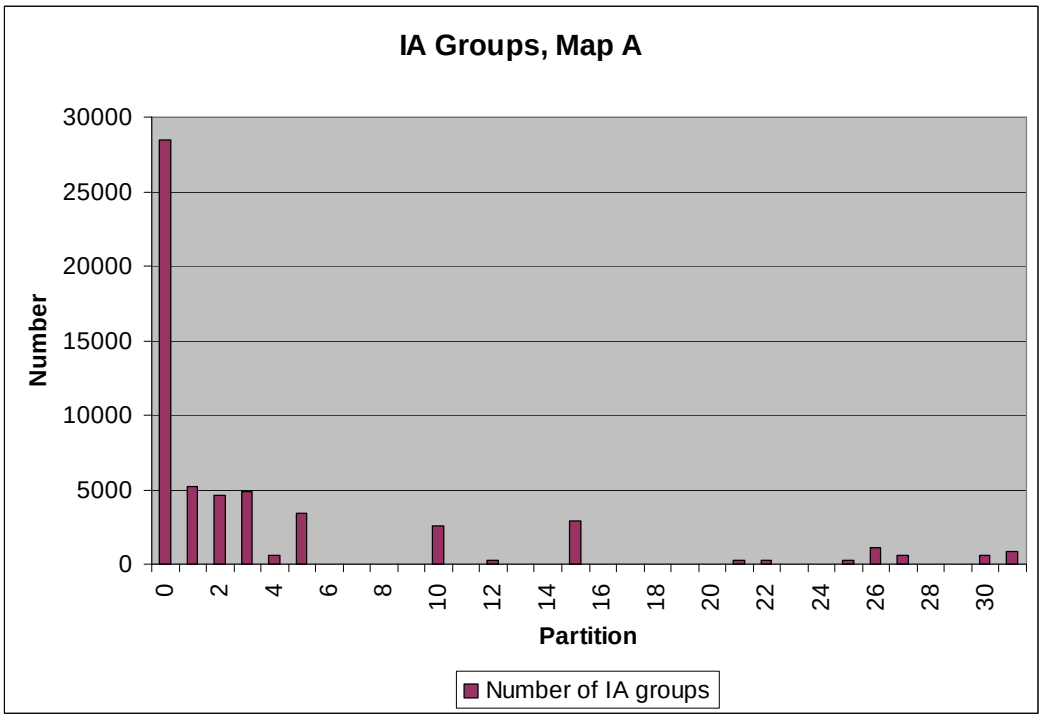

Figure 29 IA groups, Map A

Group Length	Frequency of Group Length
3	50736
7	4704
15	1568

Table 20. Frequency of group length in all partitions, Map A, $n = 6$

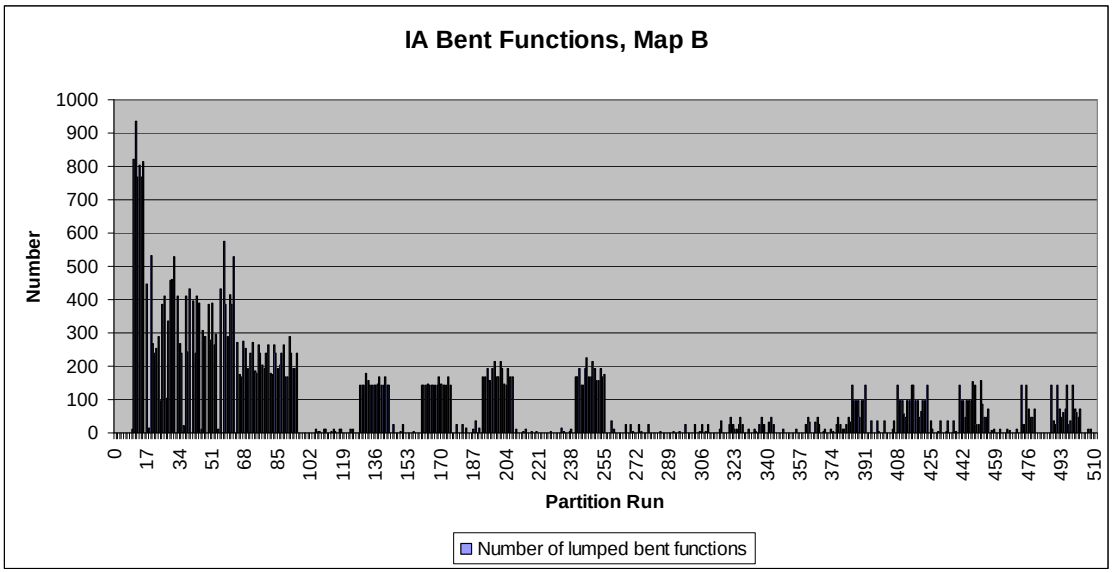

Figure 30 IA bent functions, Map B

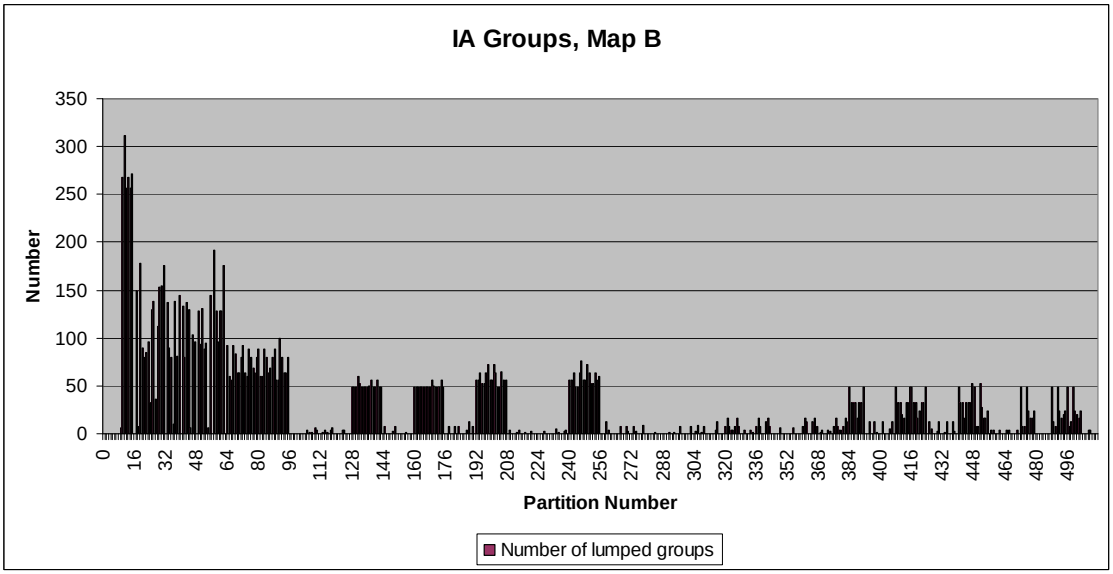

Figure 31 IA groups, Map B

Group Length	Frequency of Group Length
2	254
3	14268
4	4
5	32

Table 21. Frequency of group length in all partitions, Map B, $n = 6$

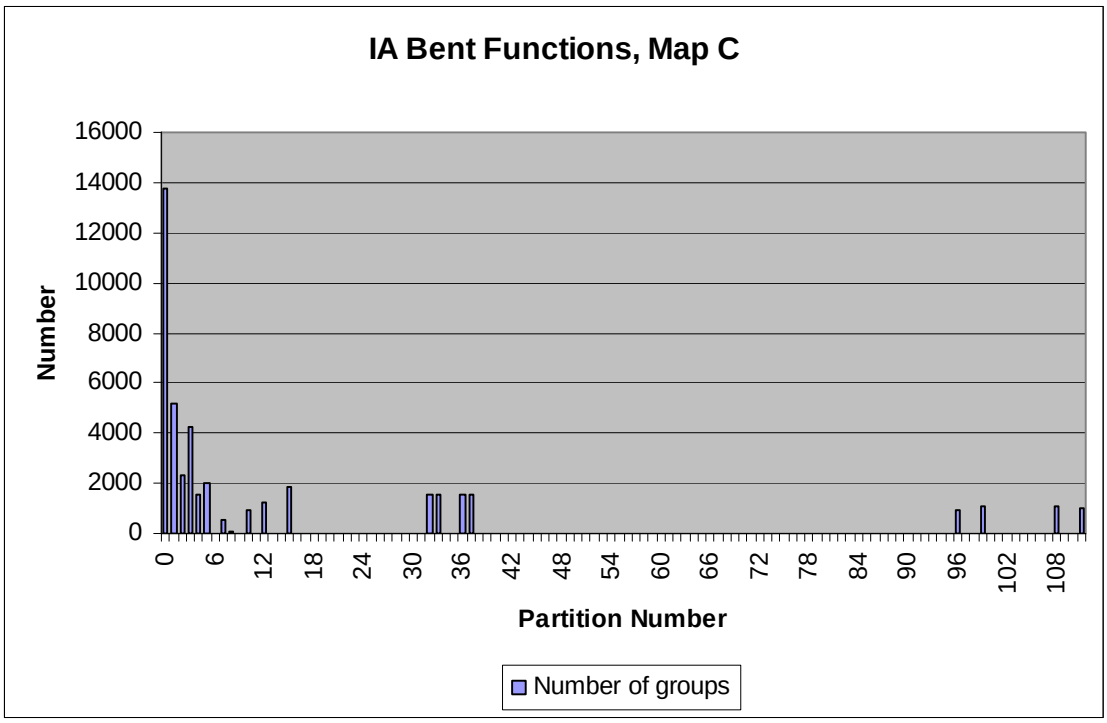

Figure 32 IA bent functions, Map C

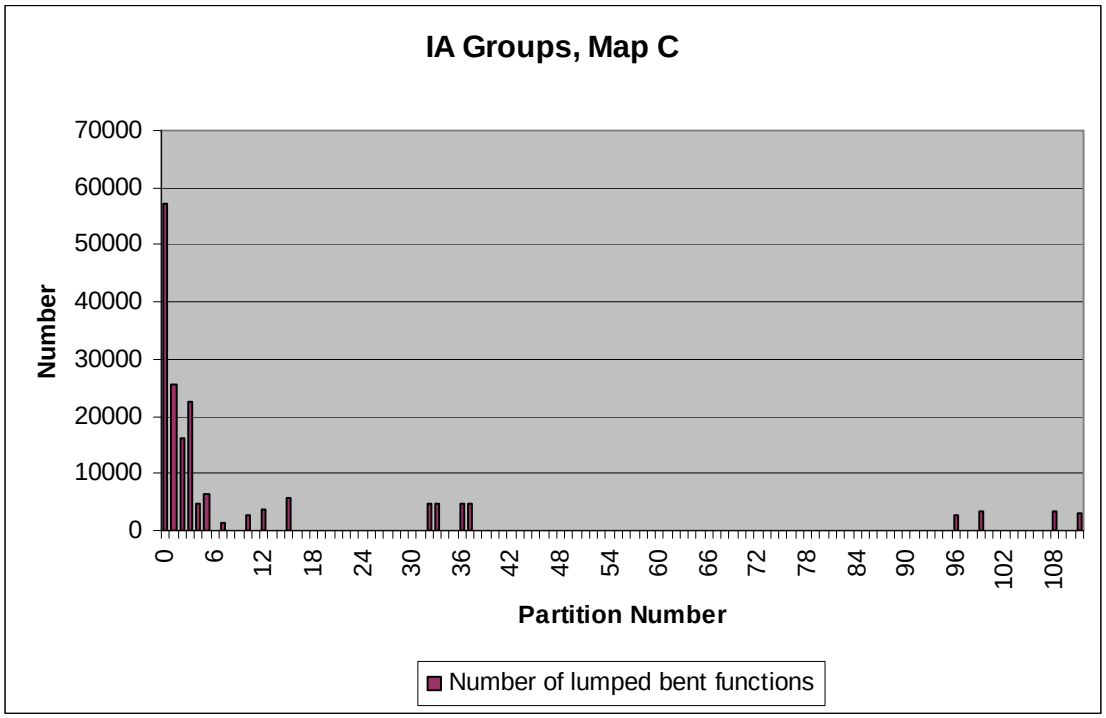

Figure 33 IA groups, Map C

Group Length	Frequency of Group Length
2	528
3	33824
4	48
5	96
7	8832
15	720
31	48

Table 22. Frequency of group length in all partitions, Map C, $n = 6$

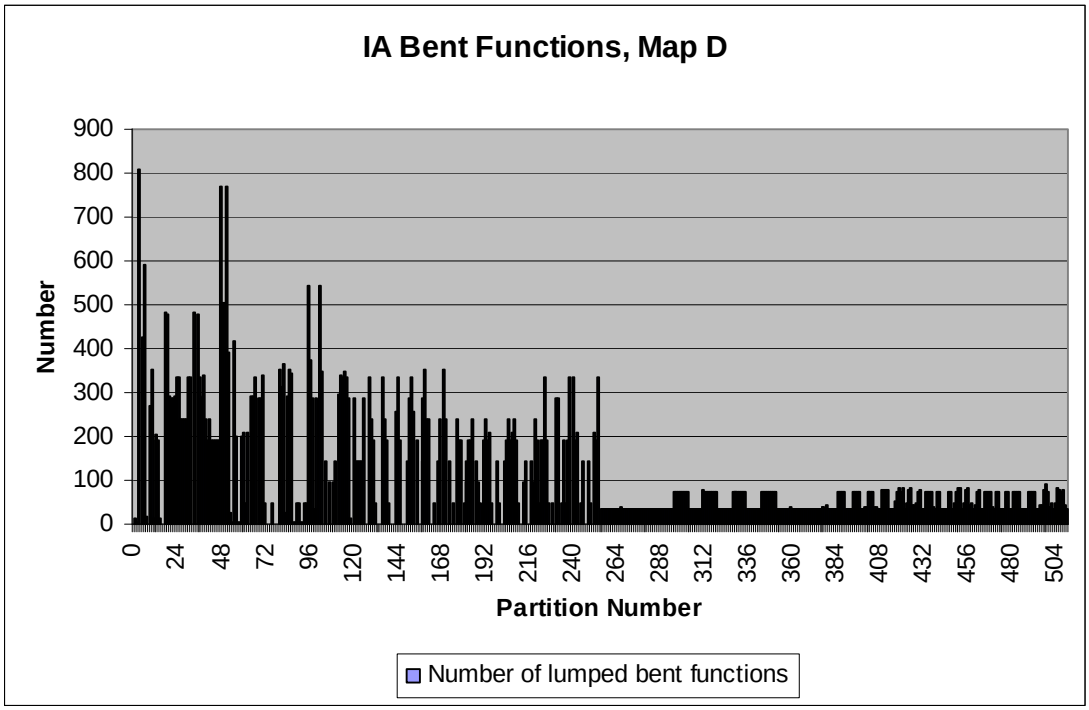

Figure 34 IA bent functions, Map D

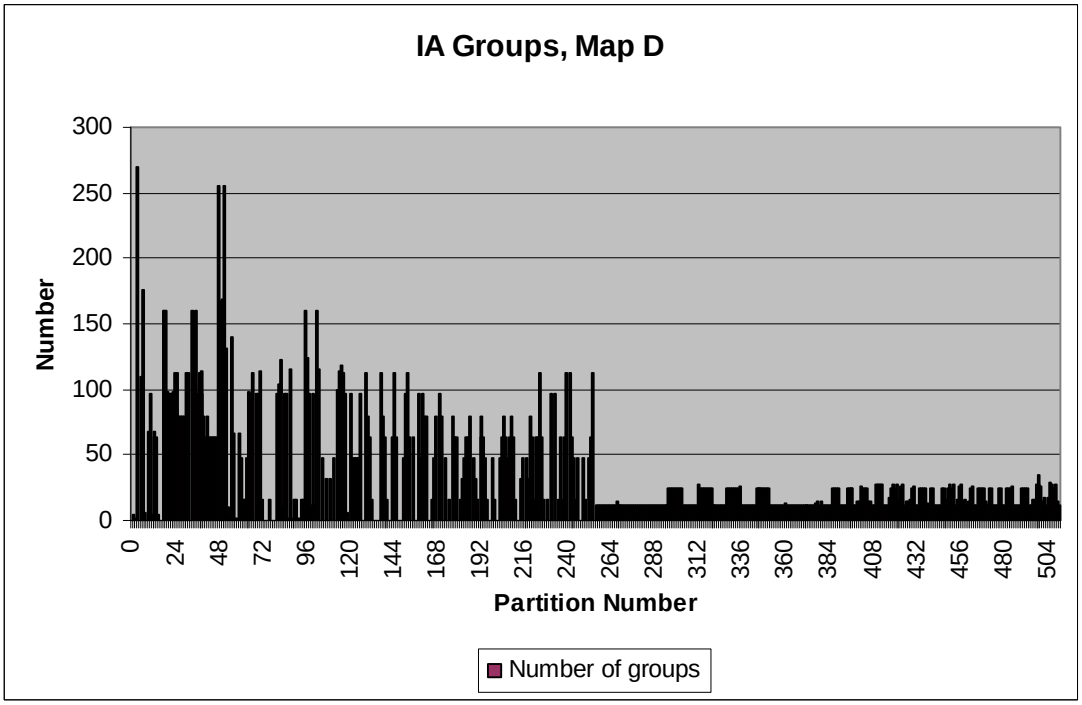

Figure 35 IA groups, Map D

Group Length	Frequency of Group Length
2	161
3	17484
4	4
7	296

Table 23. Frequency of group length in all partitions, Map D, $n = 6$

More interesting results are obtained through a random process. A C++ program was written that takes the bits that are mapped to an index and puts them into a data structure. The ordering of the elements in the data structure is then shuffled. The first element is assigned the next sequential index, and then removed from the data structure. This process continues until the data structure is empty. The Figures 36 and 37 and Table 24 show the distribution of the IA bent functions groups and their corresponding lengths.

These results show several things of interest. First, is that each partition run has at least one IA bent function group. Closer analysis shows that the minimum number of IA groups in a partition run is 8. In all of the other maps, at least one partition has no IA groups. The other item of interest is that all IA groups have a length of either 2 or 3.

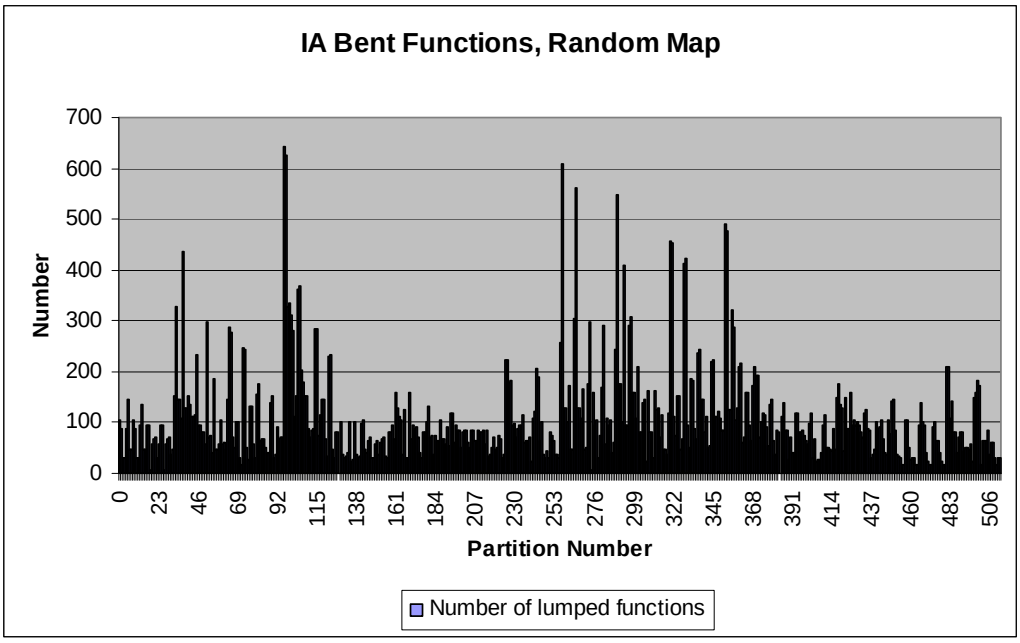

Figure 36 IA bent functions, Random Map

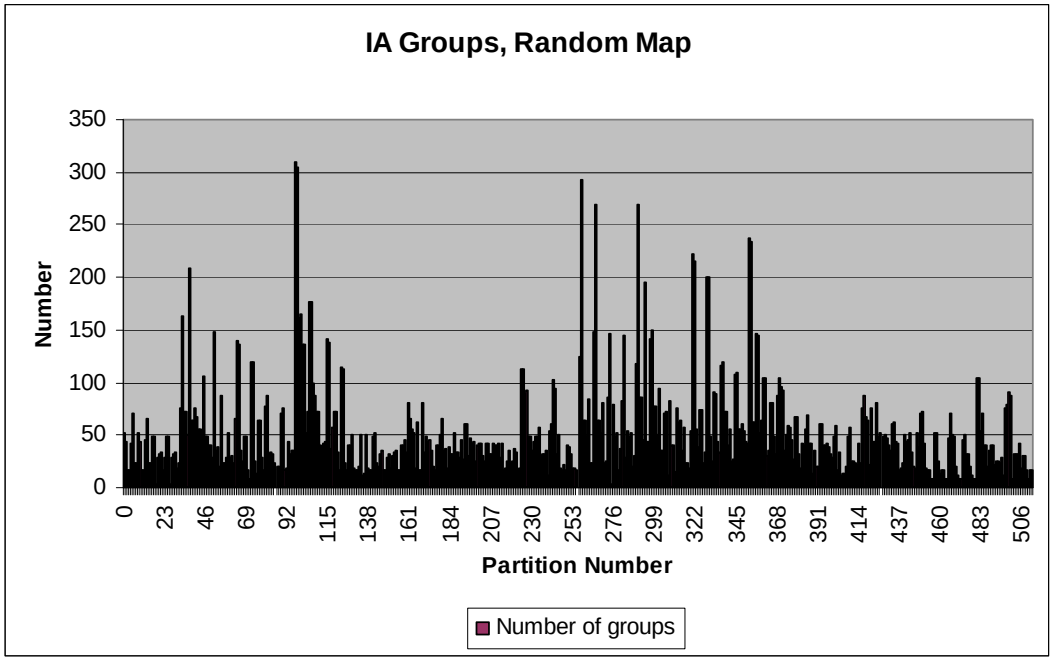

Figure 37 IA groups, Random Map

Group Length	Frequency of Group Length
2	26254
3	794

Table 24. Frequency of group length in all partitions, Random Map, $n = 6$

An analysis of this data yields two conclusions. The first is that a future GA may be able to incorporate this process in finding bent functions. Suppose the chromosome is the random index mapping to the ANF representation of a function. If mutation is defined as the addition of a bit to this index, it is possible that one bent function could mutate into another bent function. Additionally, it may be practical to use the random index map on the index mapper in order to find a bent function on $n = 10$.

Based on the figures and tables above, the following term is introduced:

$$IA concentration = \#of\ IA groups\ x \frac{\#\ partitions\ with\ IA bent\ functions}{\#partitions}$$

Table 25 shows that the IA concentration is highest for the random map. This is of interest on how to choose the mapping for a GA that implements a function's ANF as a chromosome. If mutation is implemented as adding 1 to the index, the mutated chromosome of a bent function may also be bent.

	IA concentration
Map A	2,004.19
Map B	9,070.32
Map C	2,067.00
Map D	15,701.88
Random	27,048.00

Table 25. IA concentration on $n = 6$

Figures 38 to 42 show the bent function distribution per partition for the five different maps use. The first item of note is in Figures 38 and 40. In each of those tests, the first partition contains considerably more bent functions than the others. The second item is visible in the figures, and verified in Table 26. Table 26 shows the standard deviation of the number of bent functions per partition. The random distribution shows that it may be more practical to find bent functions using the random map when examining problems with more variables. This is based on the observation that as the number of variables goes up; the scarcity of bent functions goes up even more. It is proposed that since the random map has the lowest standard deviation, the existence of a bent function in a partition for $n = 8$ or $n = 10$ is more probable.

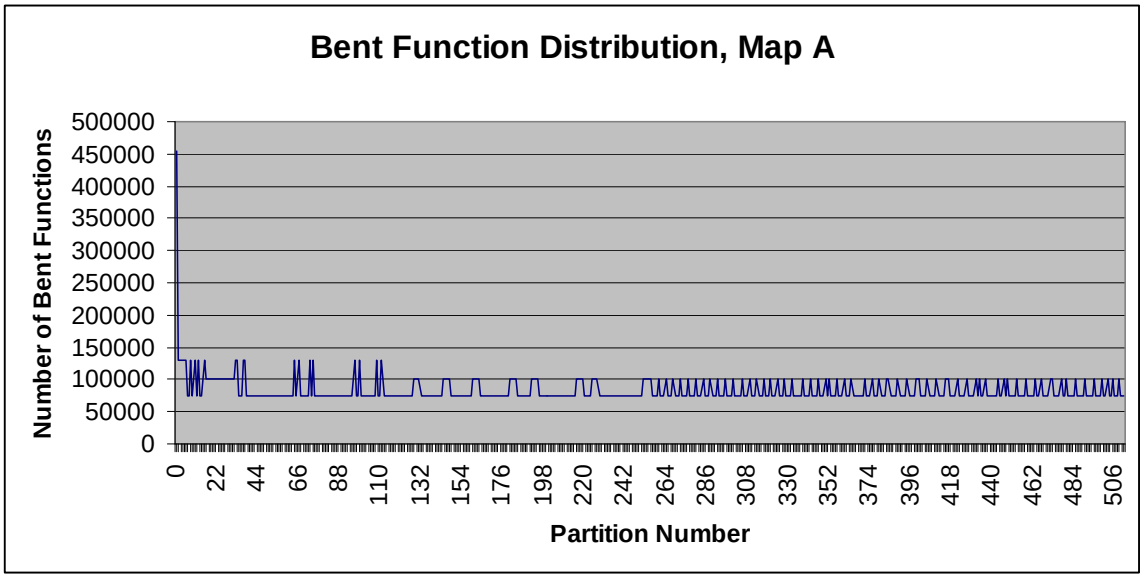

Figure 38 Bent function distribution, Map A

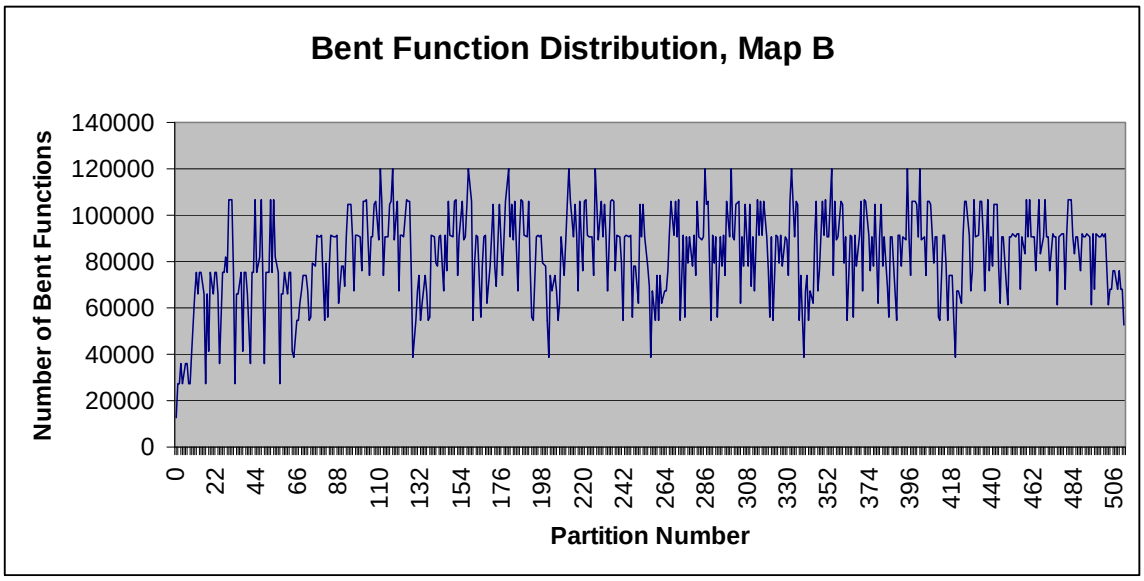

Figure 39 Bent function distribution Map B

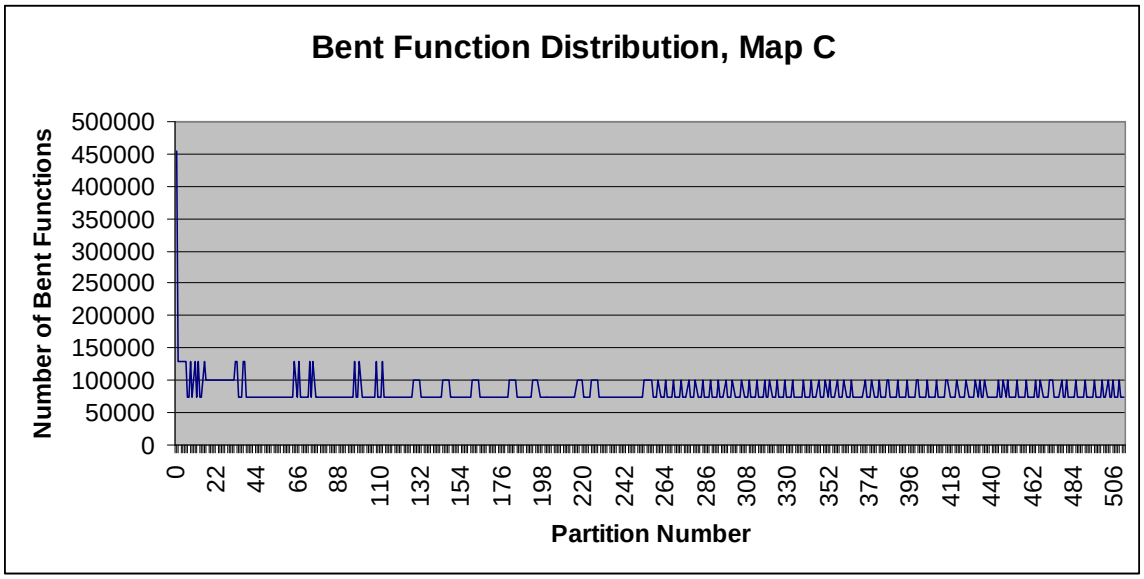

Figure 40 Bent function distribution, Map C

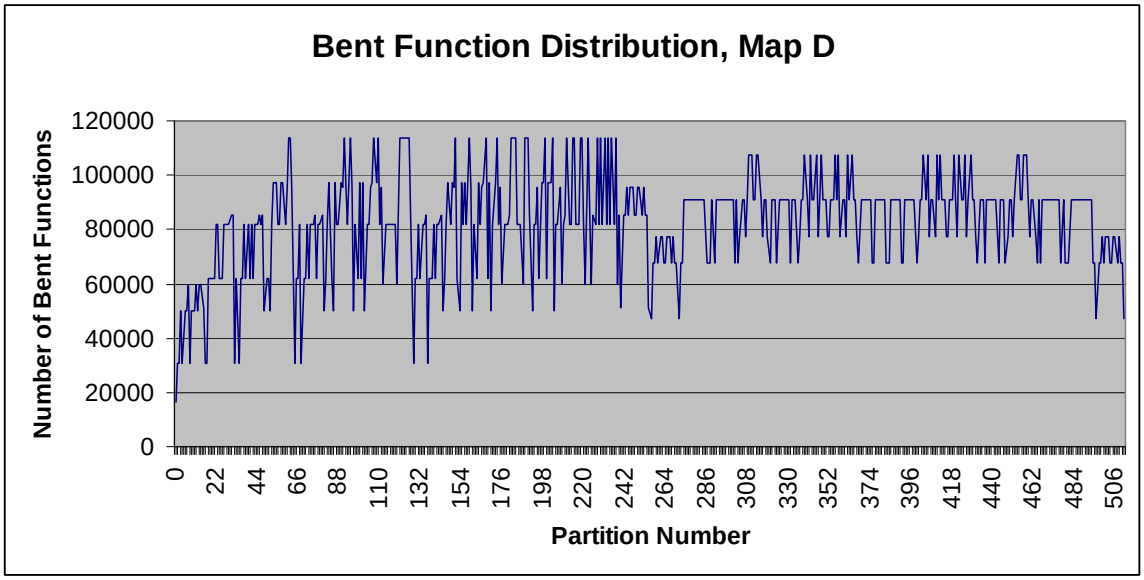

Figure 41 Bent function distribution, Map D

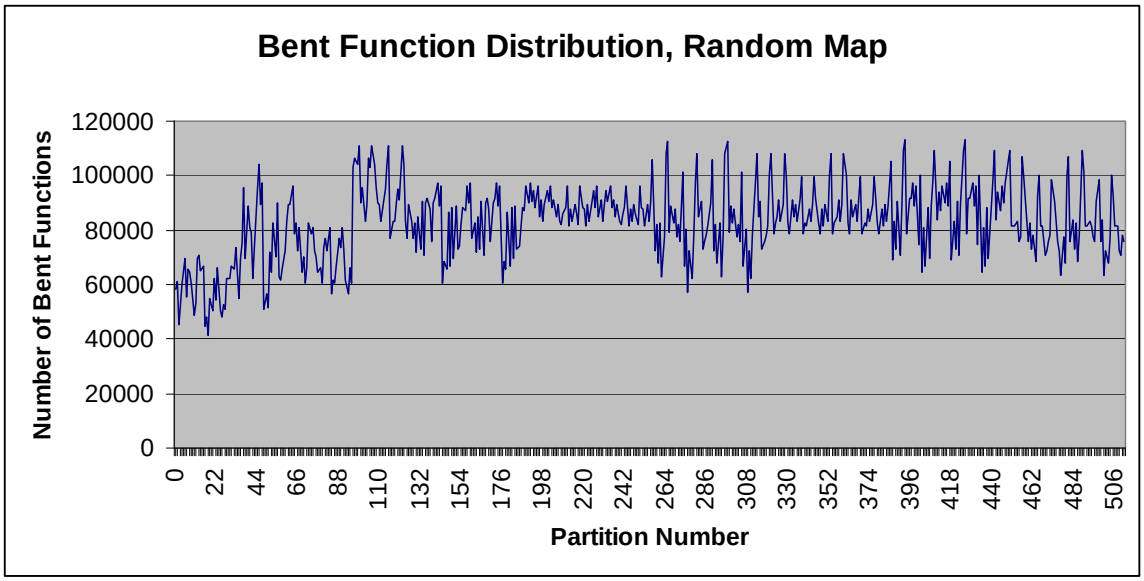

Figure 42 Bent function distribution, Random Map

	Standard deviation
Map A	22,204.96
Map B	19,339.71
Map C	22,204.96
Map D	17,993.37
Random	13,530.55

Table 26. Bent function standard deviation per partition

B. BITSTUFFING

From the weight property of bent functions, we know that a bent function has one of two predetermined number of ones in it, for a given number of variables. Because of this, we can construct a function with the appropriate number of ones in it, with the miniscule hope that it is bent. This process shall be known as bitstuffing. Although there are several properties of bent functions, that have not been discussed, that can improve the probability construction of a bent function, this discussion will focus on the ROTS functions.

1. The Ones Hypothesis

Recall that there are eight 4-variable ROTS bent functions. Each of the eight sub-tables of Table 27 represent all of the 4-variable bent functions that can be created from the 4-variable ROTS bent functions by exclusive ORing a ROTS bent function with an affine function. Note that the resulting function is bent, but not necessarily ROTS. The first sub column represents a function that has been determined to be bent and ROTS through a brute force enumeration of the search space. The second sub column is an affine function. The final sub column is the result of the exclusive ORing of the candidate ROTS bent function and that particular affine function.

Fn	Afn	Result	Fn	Afn	Result	Fn	Afn	Result
0536	0000	0536	7EE8	FFFF	8117	E881	0000	E881
0536	5555	5063	7EE8	AAAA	D442	E881	AAAA	422B
0536	3333	3605	7EE8	CCCC	B224	E881	CCCC	244D
0536	6666	6350	7EE8	6666	188E	E881	9999	7118
0536	0F0F	0A39	7EE8	F0F0	8E18	E881	F0F0	1871
0536	A5A5	A093	7EE8	5A5A	24B2	E881	A5A5	4D24
0536	3C3C	390A	7EE8	3C3C	42D4	E881	C3C3	2B42
0536	9696	93A0	7EE8	6969	1781	E881	6969	81E8
0536	00FF	05C9	7EE8	FF00	81E8	E881	FF00	1781
0536	55AA	509C	7EE8	55AA	2B42	E881	AA55	42D4
0536	CC33	C905	7EE8	33CC	4D24	E881	CC33	24B2
0536	9966	9C50	7EE8	6699	1871	E881	6699	8E18
0536	0FF0	0AC6	7EE8	0FF0	7118	E881	F00F	188E
0536	A55A	A06C	7EE8	5AA5	244D	E881	5AA5	B224
0536	C33C	C60A	7EE8	3CC3	422B	E881	3CC3	D442
0536	6996	6CA0	7EE8	9669	E881	E881	6996	8117
177E	FFFF	E881	8117	0000	8117	FAC9	FFFF	0536
177E	5555	422B	8117	5555	D442	FAC9	AAAA	5063
177E	3333	244D	8117	3333	B224	FAC9	CCCC	3605
177E	6666	7118	8117	9999	188E	FAC9	9999	6350
177E	0F0F	1871	8117	0F0F	8E18	FAC9	F0F0	0A39
177E	5A5A	4D24	8117	A5A5	24B2	FAC9	5A5A	A093
177E	3C3C	2B42	8117	C3C3	42D4	FAC9	C3C3	390A
177E	9696	81E8	8117	9696	1781	FAC9	6969	93A0
177E	00FF	1781	8117	00FF	81E8	FAC9	FF00	05C9
177E	55AA	42D4	8117	AA55	2B42	FAC9	AA55	509C
177E	33CC	24B2	8117	CC33	4D24	FAC9	33CC	C905
177E	9966	8E18	8117	9966	1871	FAC9	6699	9C50
177E	0FF0	188E	8117	F00F	7118	FAC9	F00F	0AC6
177E	A55A	B224	8117	A55A	244D	FAC9	5AA5	A06C
177E	C33C	D442	8117	C33C	422B	FAC9	3CC3	C60A
177E	9669	8117	8117	6996	E881	FAC9	9669	6CA0
6CA0	0000	6CA0	935F	FFFF	6CA0			
6CA0	AAAA	C60A	935F	5555	C60A			
6CA0	CCCC	A06C	935F	3333	A06C			
6CA0	6666	0AC6	935F	9999	0AC6			
6CA0	F0F0	9C50	935F	0F0F	9C50			
6CA0	A5A5	C905	935F	5A5A	C905			
6CA0	3C3C	509C	935F	C3C3	509C			
6CA0	6969	05C9	935F	9696	05C9			
6CA0	FF00	93A0	935F	00FF	93A0			
6CA0	55AA	390A	935F	AA55	390A			
6CA0	CC33	A093	935F	33CC	A093			
6CA0	6699	0A39	935F	9966	0A39			
6CA0	0FF0	6350	935F	F00F	6350			
6CA0	5AA5	3605	935F	A55A	3605			
6CA0	3CC3	5063	935F	C33C	5063			
6CA0	6996	0536	935F	9669	0536			

Table 27. ROTS bent functions on $n = 4$

This implies that a sieve can be easily constructed to help search for the bent functions. Furthermore, this sieve can be constructed to target ROTS bent functions. Suppose you have a function you would like to determine is bent. An initial step could be to count the number of ones in it, and for the case of n=4, see if the answer is either 6 or 10. If this is not the case, then the function is not bent. If this is the case, then it may be bent, but there is no guarantee that it is bent. The remainder of this discussion will focus on the case of a bent function containing 6 ones.

The goal is to determine if a ROTS function has 6 ones. To start, the function that maps the rotationally symmetric index (RSI) to the truth table must be examined. One lab during EC4820 involved the creation of a C program that determined the rotationally symmetric index function. This paper will only focus on those results. The below table demonstrates how the 6-bit RSI is mapped into a 16-bit truth table.

To create the 16-bit number from the RSI, each of the 6-bits of the RSI is examined one at a time. The RSI and the 16-bit truth table are formatted with the bits laid down with the 0 bit being on the right. The value of the i^{th} bit of the RSI is then mapped into the 16-bit number according to the below table. This means that the value of bits 8, 4, 2 and 1 in the expanded number are the same and equal to the value of bit 1 in the RSI.

Next, the number of ways to create a 16-bit number from the RSI was examined. This was done via a histogram on the bin frequency of the RSI in an Excel spreadsheet. Table 28 shows these results:

Source bit	Frequency
0	1
1	4
2	4
3	2
4	4
5	1

Table 28. RSI histogram, n=4

In this table, a value from each of the 6-bits in the RSI is used the number of times listed in the frequency column. From this, it can be shown that there are six ways to get a number with 6 ones. To get a number that has 6 ones in it, each source bit must be set high until the frequency total is 6. Thus, the source bit combinations that are required to get 6 ones is 015, 025, 045, 13, 23 and 34. However, since there are only 4 ROTS bent functions on $n = 4$, we know that 2 of them must be eliminated. Table 29 shows the 6 ways to get 6 ones in the 16-bit format. The two functions that are not bent are highlighted in red. Because not all functions generated through this process are ROTS bent function, it is only a sieve to construct ROTS bent functions.

In Table 29, the "selected source bit" column shows which bits in the RSI must be high to get 6 ones in the 16-bit format. The next 4 columns labeled 15, 11, 7 and 3 represent each of the 4 nibbles in the 16-bit number. Each column is labeled according to the MSB in each nibble. The final column is the 16-bit hexadecimal representation of the number being constructed. The bolded red numbers indicate RS functions that have 6 ones but are not bent. This can easily be proved by finding an affine value that, when exclusive ORed with the function, produces a result with less than 6 ones. For $0x9249$, the affine value is $0x9669$ ($0x9249 \oplus 0x9669 = 0x0420 = 0000\,0100\,0010\,0000_2$) and is determined through a brute force enumeration of the affine functions. For $0x1668$, the

affine value is also $0x9669$ ($0x1668 \oplus 0x9669 = 0x8001 = 1000\,0000\,0000\,0001_2$). The bent functions are now known to be $0x536, 0x6CA0, 0x8117$ *and* $0xE881$ through enumeration.

	Binary representation				
Selected source bit	15	11	7	3	Hex format
015	1000	0001	0001	0111	8117
025	**1001**	**0010**	**0100**	**1001**	**9249**
045	1110	1000	1000	0001	E881
13	0000	0101	0011	0110	0536
23	**0001**	**0110**	**0110**	**1000**	**1668**
34	0110	1100	1010	0000	6CA0

Table 29. RSI functions with 6 ones

This same concept can be used to produce a sieve for larger cases of n. The case implemented for this thesis was for n=6. This results in a 14-bit RSI being mapped into a 64-bit number. The RSI mapping function has been previously discussed under the guise of the 28 pound packing problem discussed in III.B.

2. Execution of a GA on $n = 6$

The GA has been implemented on the SRC-6, as previously described in this thesis. The results of the data trends from the SRC-6 are discussed below. As will be described, different parameters of the GA were altered in order to see how the results varied. However, for the sake of consistency throughout all of the experiments discussed, the same set of seeds is used for the LFSRs. Additionally, all of the ROMs were addressed in a sequential manner, versus the CRC addressing method previously described.

The following terms will be used throughout the discussion of the GA results. Since the GA implements the ones' count hypothesis, it is a sieve for bent functions. What that means is that a "fit" chromosome from the GA, meaning it has the maximum fitness value of 240, may be or may not be bent. The term "percent fit" describes how many of the chromosomes out of all of the generations have a fitness of 240. If a chromosome has a fitness value of 240, and its corresponding truth table is a bent function, then it is referred to as a "chromosome yielding a bent function". The "yield" is the percentage of all chromosomes with a fitness of 240 whose TT yields a bent function. Bent functions are only counted once regardless of how many times they generated through the chromosomes, and are thus referred to as "unique". Unless specified otherwise, the minimum fitness value to prevent chromosome replacement is 150.

Several methods were used to implement the crosscode. Unless stated otherwise, the crosscode of $0x1F$ is used. This specifies the boundary between the bins corresponding to a weight of 6, and all other bins. This is a single point crossover. Later, two types of random crosscodes are tested. The first type is one that has the crossover occur at one place within the chromosome. The other type allows for multiple crossover points. In all cases, the same crosscode is used for all elements in a particular generation. In some cases, described in detail later, each generation has its own crosscode.

The first set of tests shows effects of changing the minimum fitness value. There are essentially three interesting things to look at in the results of when the minimum fitness value is changed. The first is when the minimum fitness value is 0, the GA behaves more like a GA in which new chromosomes are only introduced through crossover. Conversely, when the minimum fitness value is 240 the GA behaves like a brute force search, albeit with some "genetic" aspects to the brute force. Finally, the remaining interesting point is where the GA seems to change its success rate. This first set of tests was run under two generation lengths, 512 and 2,000. The former choice is based on 16 chromosomes per generation for 512 generations yields 8,192 chromosomes examined. This is half of the total chromosomes in the entire search space. The later choice is an arbitrary larger value. Figure 43 shows the number of unique bent functions found, while Figure 44 shows the number of chromosomes yielding a bent function.

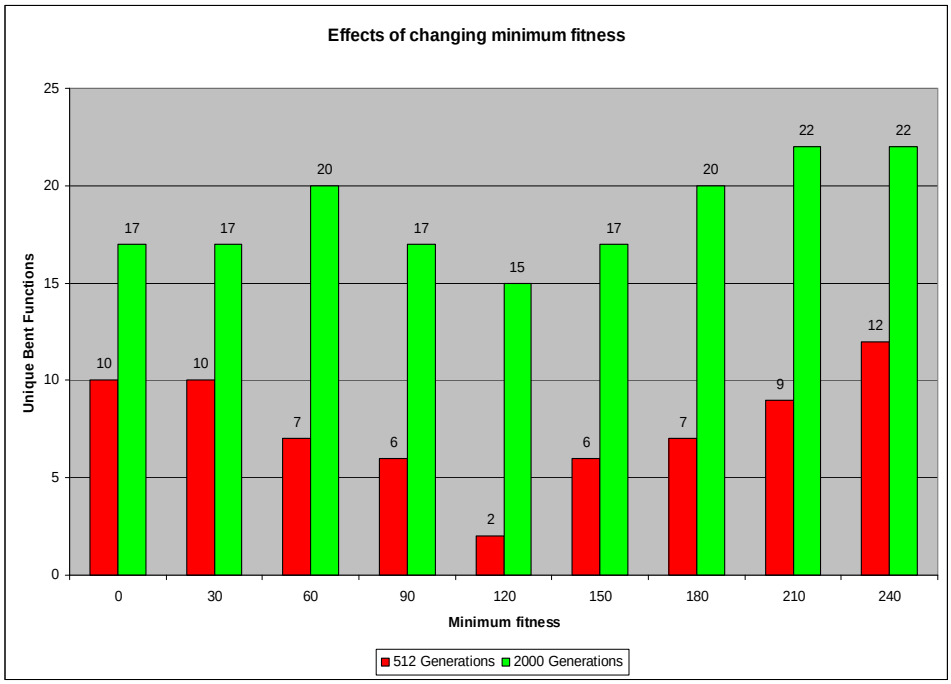

Figure 43 Unique bent functions

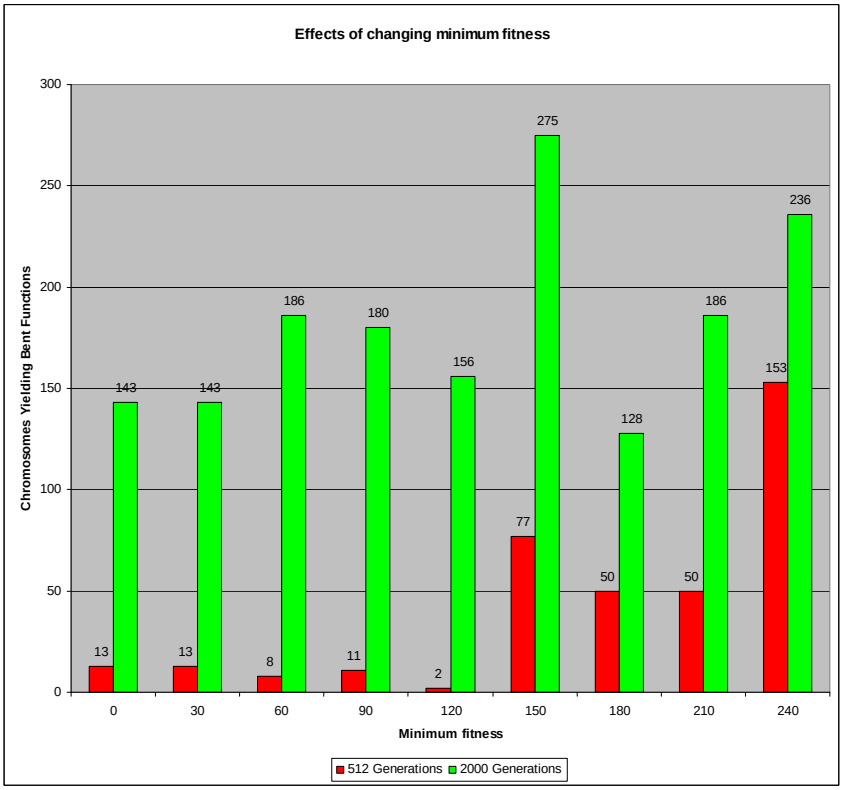

Figure 44 Chromosomes yielding bent functions

Of greatest interest here is the distinct jump in the number of chromosomes yielding a bent function between the minimum fitness of 120 and 150 as seen on the 512 generation test. This result is also shown in Figures 45 and 46, which show the percent fit and the yield for these tests. Because of these results, the minimum fitness of 150 was chosen as the minimum fitness of subsequent testing.

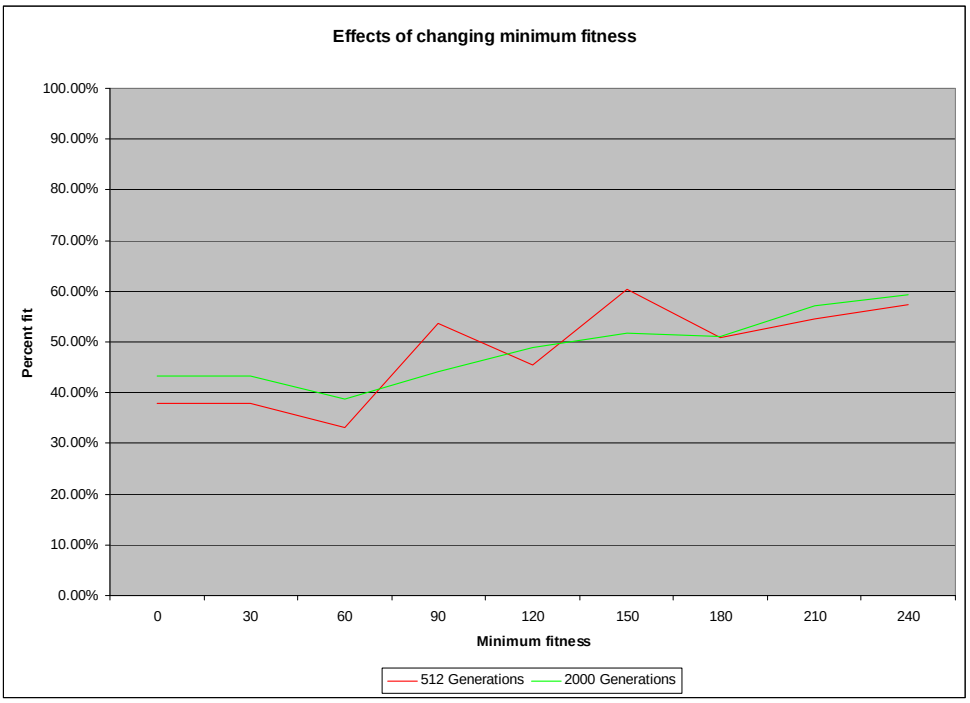

Figure 45 Percent fit versus minimum fitness

Figure 46 Yield versus minimum fitness

Figures 47 and 48 show the effect of changing the crossover point on the number of chromosomes yielding bent functions, and also the number of unique bent functions located. There are five sets of tests in this experiment, with each test running for 512 generations. The first test is the control test, with a crossover point of $0x1F$ as described with Figures 43-45. The second test uses a multi-point crossover with code $0xED0$. Multi-point crossover means that instead of one point determining where the chromosomes are split and recombined, there are several points. For example, the multi-point crosscode $0xED0 = 1110\_1101\_0000_2$ means that the bits 4, 6, 7, 9, 10 and 11 will crossover, where bit 0 is the LSB. The third test is a single point crossover with crosscode $0x7$. These two crosscodes were used in all generations. The final two test sets each have a unique crosscode for each generation. In the first case, the crosscode is a multi-point, and in the last case, it is a single point crossover. The most obvious piece of information gained from this set of experiments is that, with the current design of the chromosome, and fitness function, the success of the GA is definitely dependant on the

crossover code. This can be seen in two places. The first is on the case where the minimum fitness value is 120. By the design of the fitness function, 120 is a key point since it represents that 1 of the 2 subproblems has been "solved". The other issue is the randomly generated multi-point crosscode yields some interesting results when the minimum fitness value is 150. The correlation between the crossover code and the minimum fitness value appears to be dependant on the minimum fitness value. As it gets higher, different crosscodes yield better results. This may be due to the GA starting to resemble a brute force attack as the minimum fitness value approaches 240.

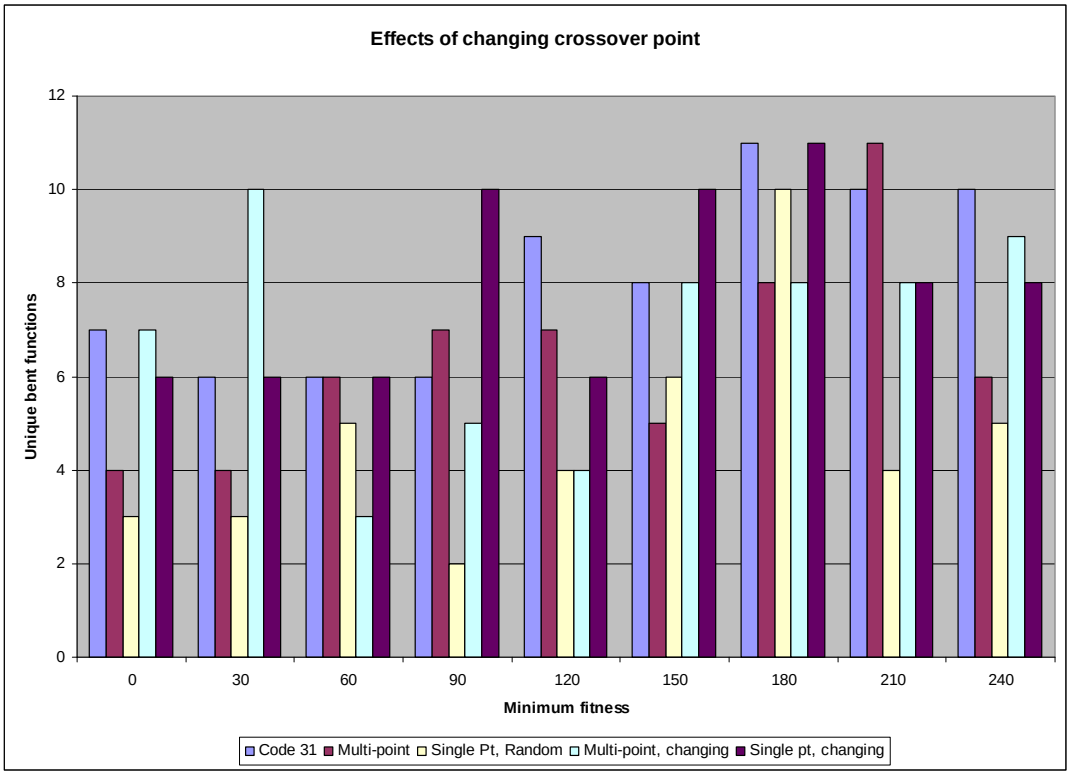

Figure 47 Effect on number of unique functions due to changing the crossover point

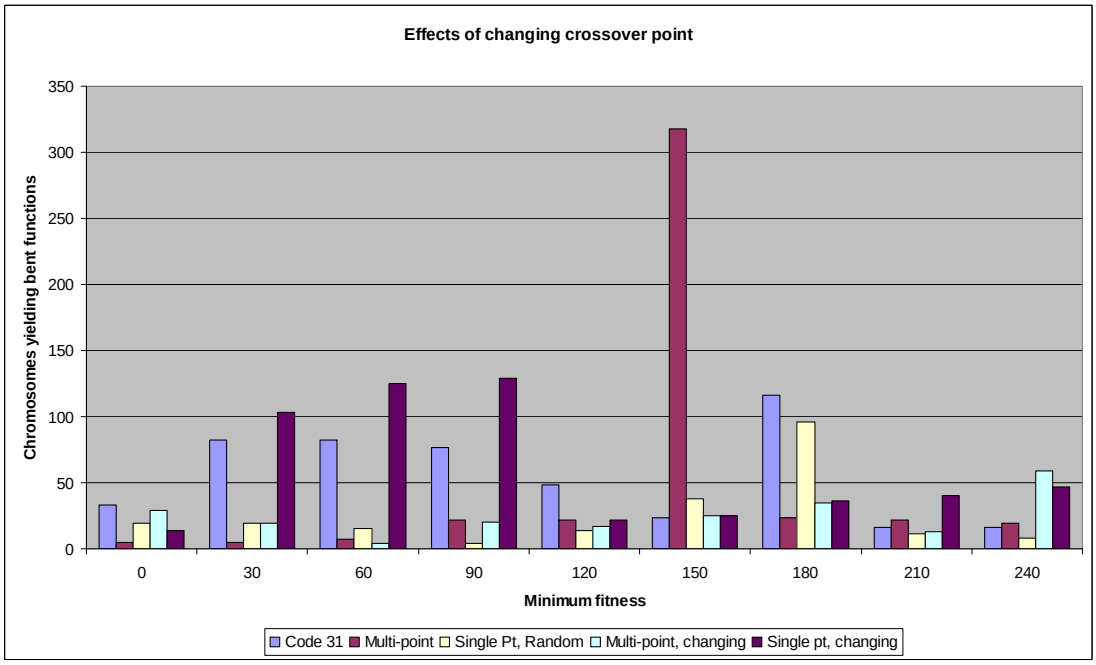

Figure 48 Effect on number of chromosomes yielding bent functions due to changing crossover point

Figures 49 and 50 show the effect of changing the crossover point has on percent fitness, and the percent yield. Overall, each of the crossover methods has a tendency to yield more fit chromosomes as the minimum fitness value rises. The yield is somewhat affected, although the values are too small to draw any definitive conclusions, with the exception of a few data points that show a marked change in the performance of the crossover point with respect to the minimum fitness value. The most notable points are shown on Figure 50. The test single point change has its percent yield drop go from 4.77% to 0.73% as the minimum fitness changes from 90 to 120. On the multipoint test, the yield goes from 0.62% to 8.87% to 0.53% as the minimum fitness value changes from 120 to 150 to 180.

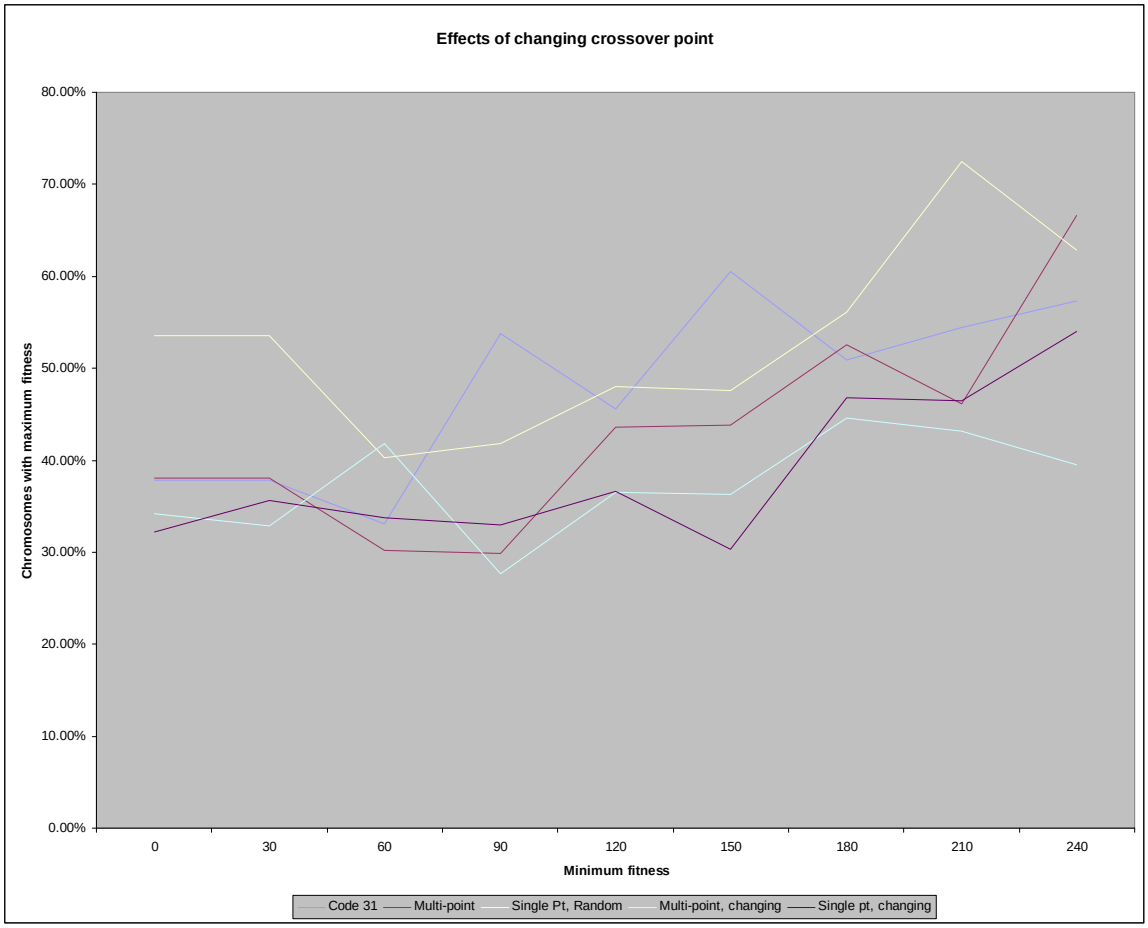

Figure 49 Percent versus minimum fitness for a changing crosscode

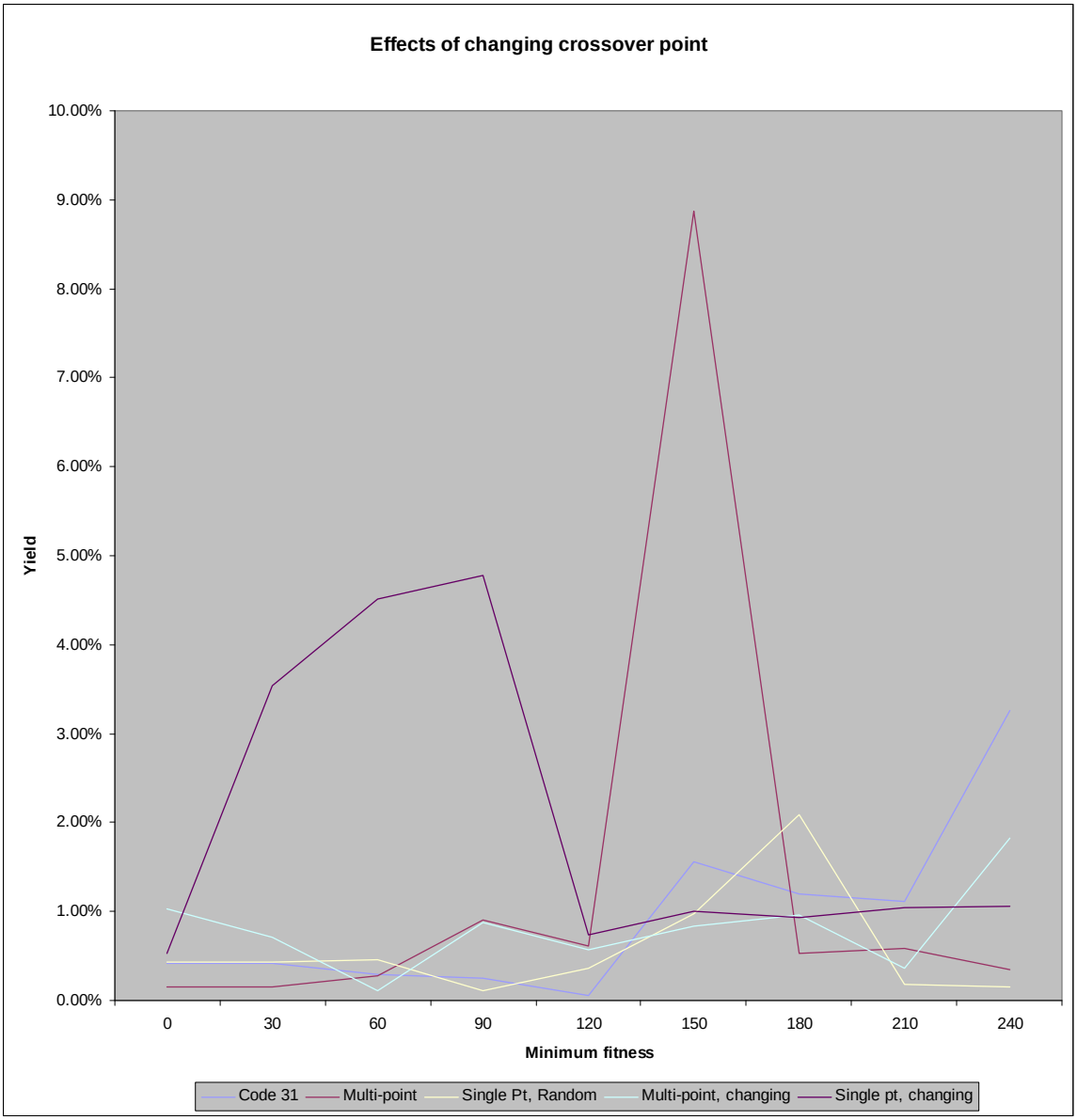

Figure 50 Yield versus minimum fitness for a changing crosscode

The final set of tests involves the effects of changing the number of generations. Again, the original crosscode of $0x1F$ is used. Three cases are examined for the minimum fitness value. Again, the minimum fitness is varied in increments of 30 from 0 to 240. Figures 51 and 52 show some interesting results. The first is that the number of unique bent functions becomes saturated very quickly. In essence, this "saturation" is actually a limit indicating that all of the ROTS bent functions on $n = 6$ that could be found were found. That statement has its basis in the fact that this GA is only searching

for ROTS functions that have 28 ones in it. This process will only locate half of the ROTS bent functions, since only half of them have 28 ones in them. Their complements, however, contain the other half of the ROTS bent functions on $n = 6$. This means that for all cases, running the GA for more than 6,000 generations does not yield any additional bent functions. Prior to that threshold, there is a linear relationship between the number of unique bent functions and the number of generations. The number of chromosome yielding bent functions behaves drastically differently, though. As the number of generations increase, there is nearly a logarithmic rise in the number of chromosomes yielding bent functions.

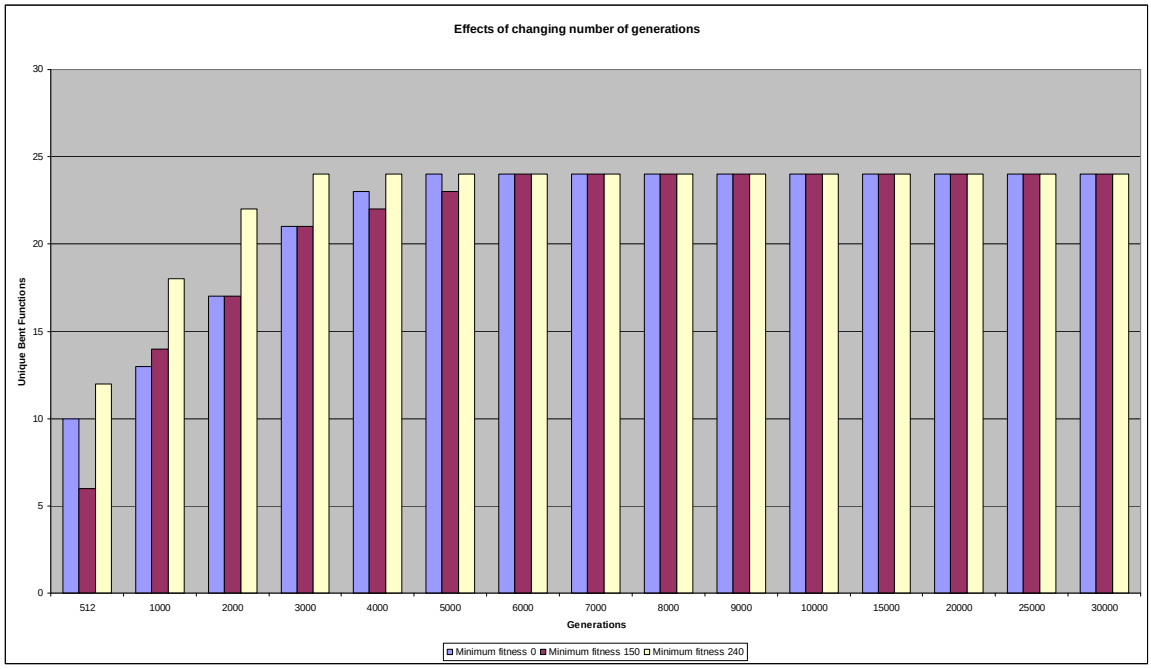

Figure 51 Unique bent functions versus changing generations

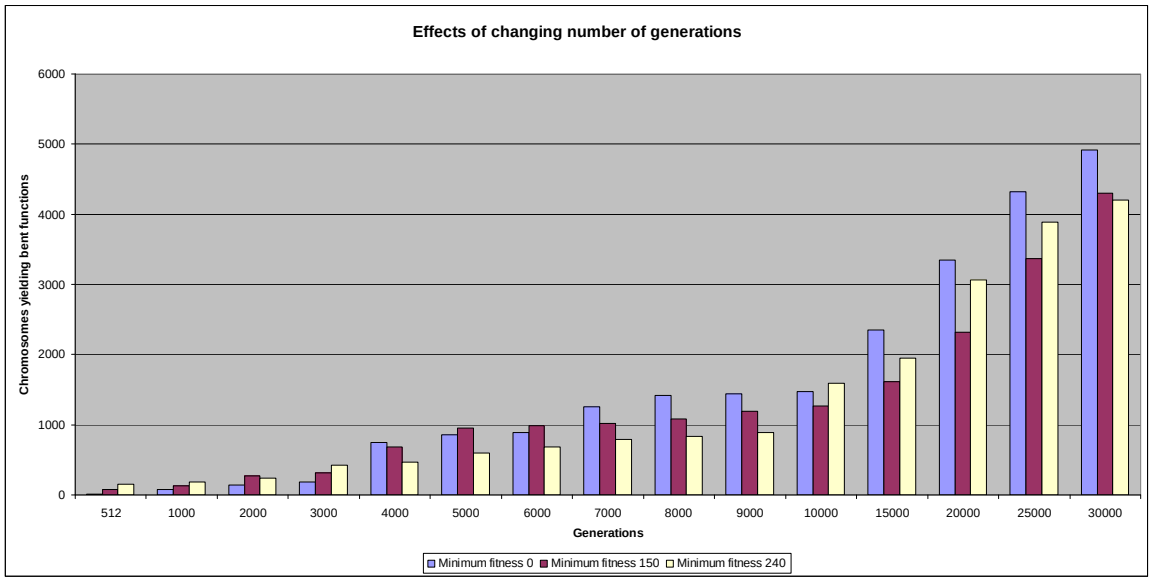

Figure 52 Chromosomes yielding bent functions versus changing generations

Figures 53 and 54 show the relationship between the percent fit and the yield. In all cases for the percent fit, there is little change once 2,000 generations have elapsed. There is an inverse relationship between the yield of the GA and the minimum fitness value versus the number of generations. For example, when the minimum fitness value is 0, the yield grows as the number of generations grows. The converse is true for a minimum fitness value of 240, as the number of generations grows, the yield diminishes. This is due to the fact that with a minimum fitness value of 240 the GA starts to emulate a brute force attack. However, since chromosomes are eliminated through selection, and chromosomes that do not have a perfect fitness are also eliminated, this case actually behaves worse than brute force. The yield for the minimum fitness value of 150 is relatively unchanged.

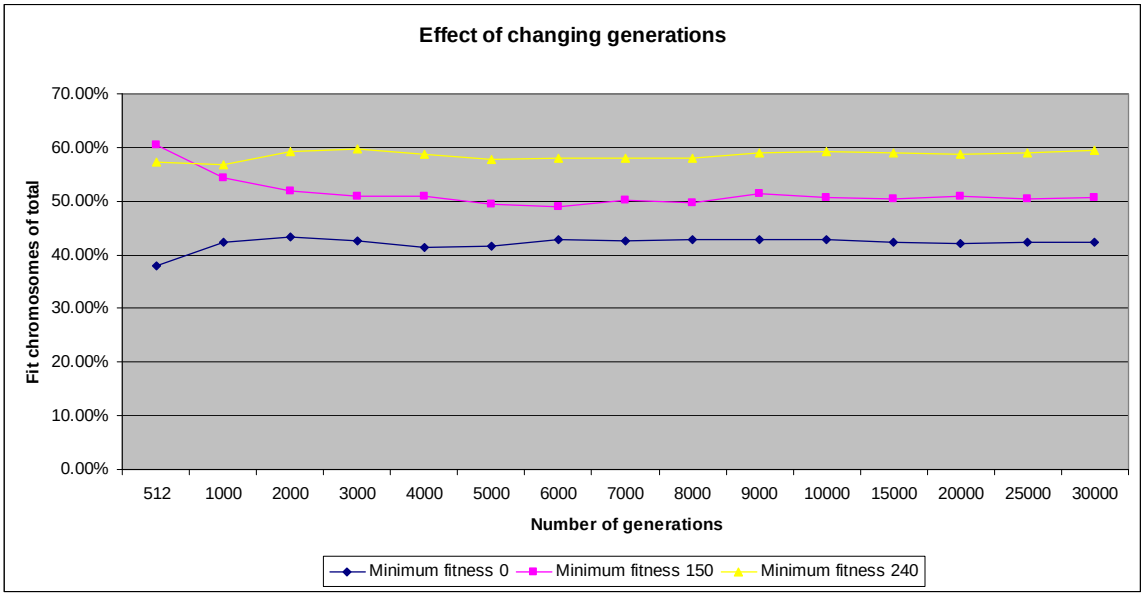

Figure 53 Percent fit versus number of generations

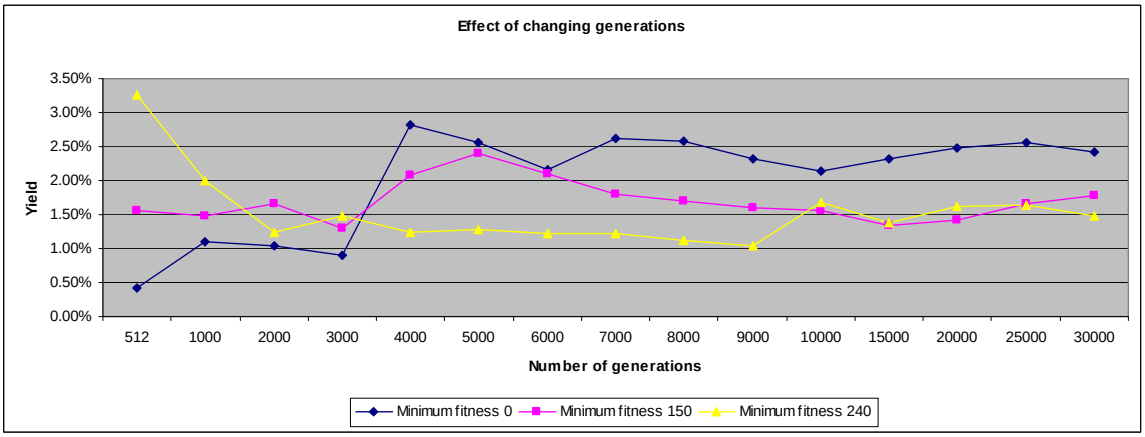

Figure 54 Yield versus number of generations

Finally, a test was conducted to determine the effectiveness of crossover on producing bent functions. This analysis is limited to the tests involving 512 generations. As more generations elapse, the LFSRs produce the set of pseudorandom numbers that rapidly covers the entire search space. This can be accomplished in as little as 1,024 generations if the LFSR seeds are strategically space, which they are not. Regardless, it becomes extremely difficult to determine if a chromosome is the resultant of it being generated by the LFSR, or by crossover.

This process starts by storing all of the values that the LFSR will generate over 512 generations into a text file. This text file is then read, and its values are stored into a C++ set data structure. This type of data structure can be viewed as a specialized vector in which no two elements have the same value. It should be noted that the underlying data structure in a set is probably not a vector; a vector is merely mentioned due to its familiar nature. Next, the chromosomes with a fitness value of 240 are loaded into a different set. Each element of the first set is searched for in the second set, and removed if found. The elements remaining in the second set after this process are those that must have been generated through crossover. It is possible that some fit chromosomes were generated through crossover that were also generated by the LFSRs, and are thus being masked. However, given the results of Figure 55, a sufficient number of fit chromosomes are being generated through crossover that further investigation on the success of crossover is not warranted.

Figure 55 shows the percentage of all fit chromosomes that could not have been produced by the LFSRs during the run of 512 generation. This figure shows that crossover is indeed being effectively used to generate fit chromosomes. To be specific, fully $1/3$ to $1/2$ of the fit chromosomes came from crossover. Actually, it cannot be determined if they came from crossover or mutation. However, by design, mutation is already sufficiently rare and ineffective that its contribution can be neglected.

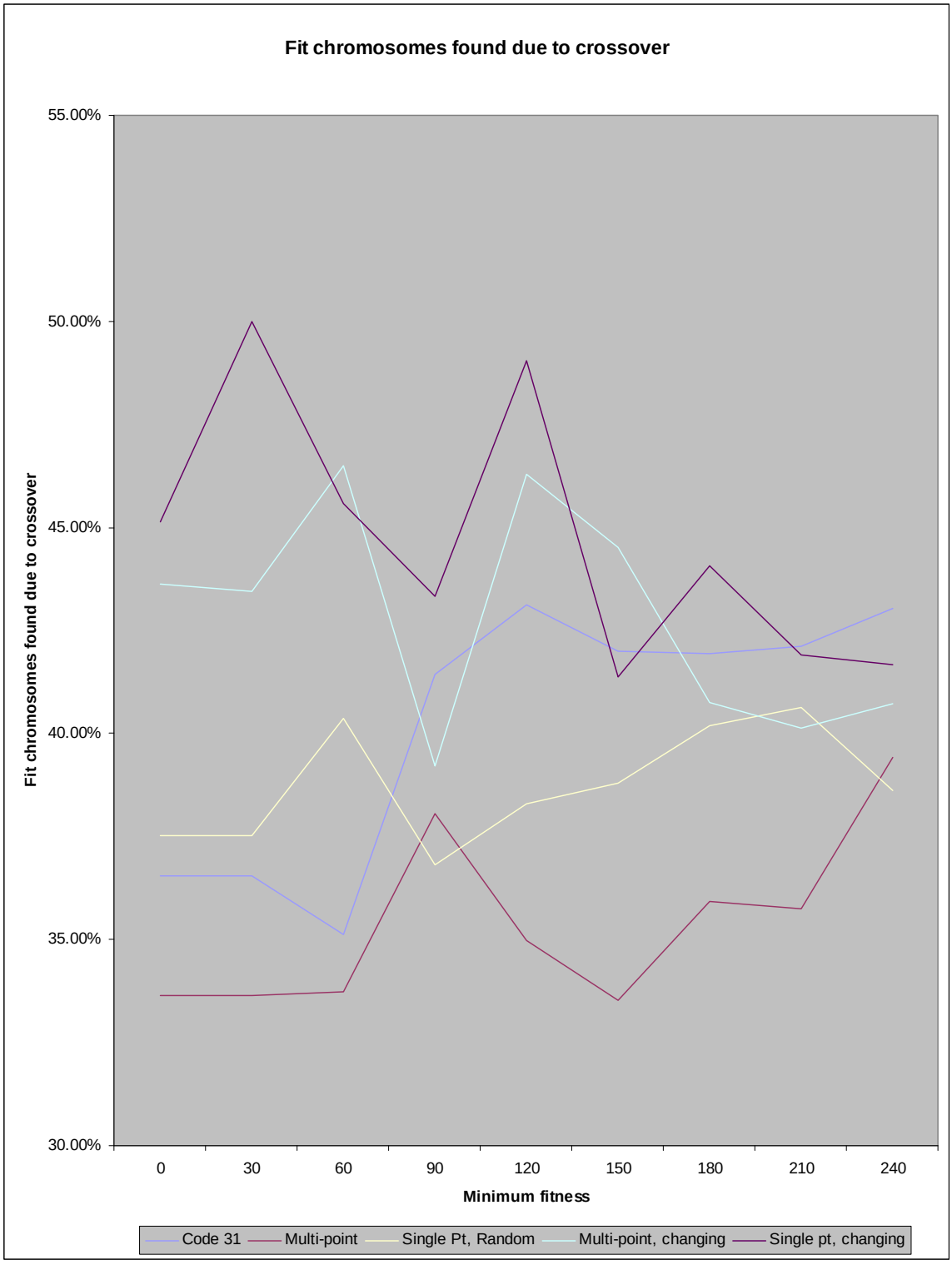

Figure 55 Fit chromosomes due to crossover

C. ROTS SEED ON $n = 4$

To assist in reviewing the bent functions on $n = 4$, a computer program was written that serves as a filter for the bent functions. The user is able to specify which of the nibbles he wishes to examine, and those bent functions that meet that requirement are shown. From this, it was observed that a given co-function exits numerous times across all of the bent functions. To further process this information, another program was written that specifically tallied all of the co-functions, and their respective frequency in both the high and the low position. These results yielded an interesting observation. There are 112 unique co-functions on $n = 4$. Each of those co-functions appears 16 times. From this it is hypothesized that bent functions of a certain number of variables can be constructed using co-functions of the same number of variables.

After an extensive comparison of the co-functions with the affine functions, the following algorithm is given to find bent functions on $n = 4$. It is proposed that this algorithm be implemented to cover larger n.

First, all of the affine functions are loaded into a C++ set data structure. Next, all of the given bent functions are loaded into a different set. The bent functions that exist during the initial execution of this algorithm are known as the bent function seed. Later discussion will show how ROTS bent functions are ideally suited for this process.

The set of bent functions is parsed, with each bent functions being broken into its two component co-functions, with the high co-functions being logically shifted to the left to align the co-functions. Each of those two co-functions are added to a co-function set. Additionally, the bits of each co-function are reversed, and those new values are added to the co-function set. For example, suppose the set contains the co-function $0x74 = 01110100_2$. The number $0x2E = 0010\ 1110_2$ is then added to the set.

Second, a nested loop is constructed. Each loop iterated through the entire co-function set. While doing so, the current co-function from each loop is exclusive ORed, and the resultant value is placed in a temporary co-function set. After the loop completes, the temporary co-function set is merged with the original co-function set.

Third, a set of proposed bent functions is produced using a manner similar to construction of the temporary co-function set. Again, a nested loop is constructed with each loop iterating through each member of the set. A proposed bent functions is constructed by assigning the co-function in the outer loop as the high co-functions and the inner loop as the low co-function.

Finally, the nonlinearity of the proposed bent functions is calculated, and a new set of bent functions is produced. This new set of bent functions provides the input to the aforementioned algorithm to again produce a set of functions to be tested for bentness. This process was implemented using a Linux shell script.

Observation:

Only two functions are required to be in the seed, provided that they are both ROTS and of a different A-class, in order to generate all 896 bent functions.

This can be accomplished in two iterations of the algorithm. Different combinations of seeds were constructed utilizing a ROTS bent function and a bent function of a different A-class. Although the combinations were not exhaustively searched, it is observed that all 896 bent functions can be generated from a given pair of seeds. Further iterations of the algorithm fail to produce additional bent functions.

In a sense, this is actually a genetic algorithm, albeit with the order of genetic operations different from those previously discussed in this thesis. Consider each co-function from the bent functions as the chromosomes. The operations of combining the co-functions with themselves, and the affine functions, could be viewed as a form of crossover. Survival of the fittest is mimicked through the nonlinearity calculation. When the nonlinearity is determined, those functions that are not bent are removed from consideration because they are not “fit”. Thus, in this case, “fitness” is determined by the nonlinearity of the function. In this test the mutation operation is not implemented. Each iteration of the algorithm is a generation. After a few number of generations, no additional information is gained.

The nonlinearity distributions are shown in Tables 56 and 57 for $n = 4$. This is to provide a comparison against the nonlinearity distributions of higher n.

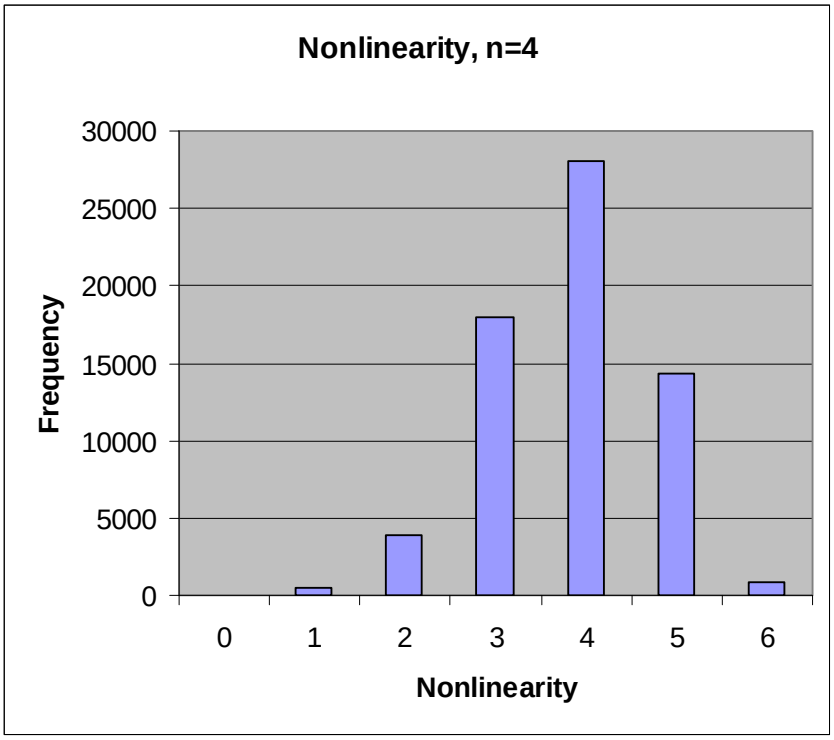

Figure 56 Nonlinearity, $n = 4$

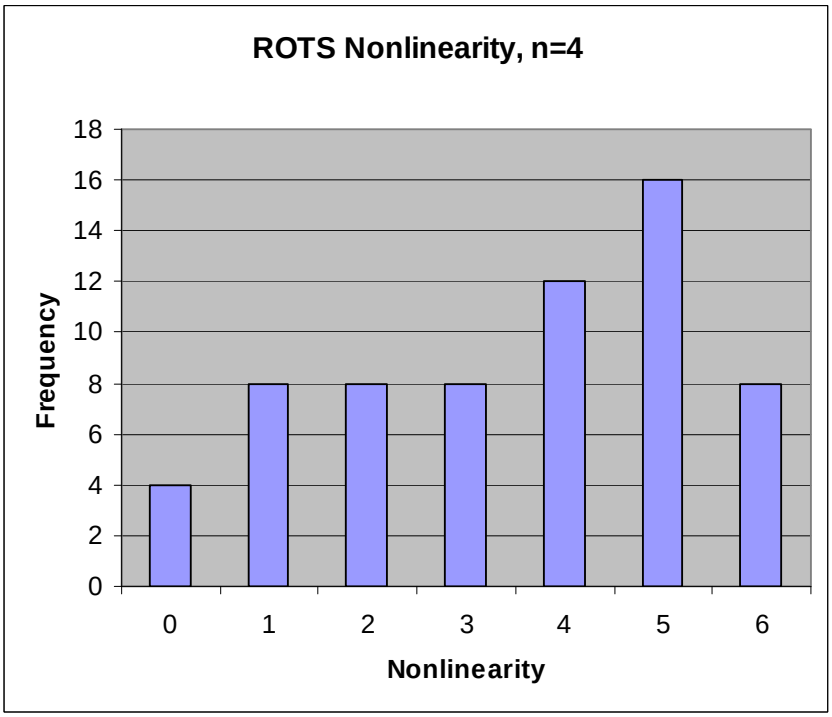

Figure 57 ROTS nonlinearity, n=4

D. BY DEGREE ON $n = 6$

As mentioned in IV.A.2, all of the bent functions were enumerated on $n = 6$ from the set of functions with degree 3 or 2. Figure 58 shows the graph for the nonlinearity distribution observed from these functions. Figure 59 shows the nonlinearity distribution for ROTS functions on $n = 6$. Of particular interest is that only the following nonlinearities were observed: 0, 8, 12, 14, 16, 18, 20, 22, 24 and 28. The distributions for nonlinearities of 0, 8, 12 and 14 are 1, 11,160, 1,749,888 and 22,855,680 respectively. The remaining nonlinearity frequencies are so large that the previous nonlinearities are not noticeable in the figure. Note that the two graphs share the same basic shape.

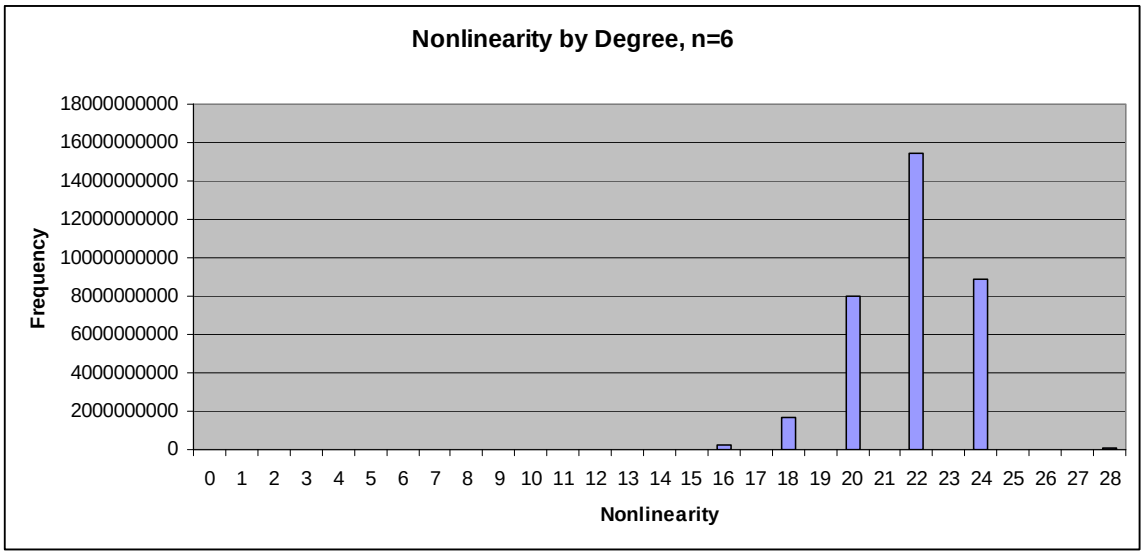

Figure 58 Nonlinearity by degree, $n = 6$

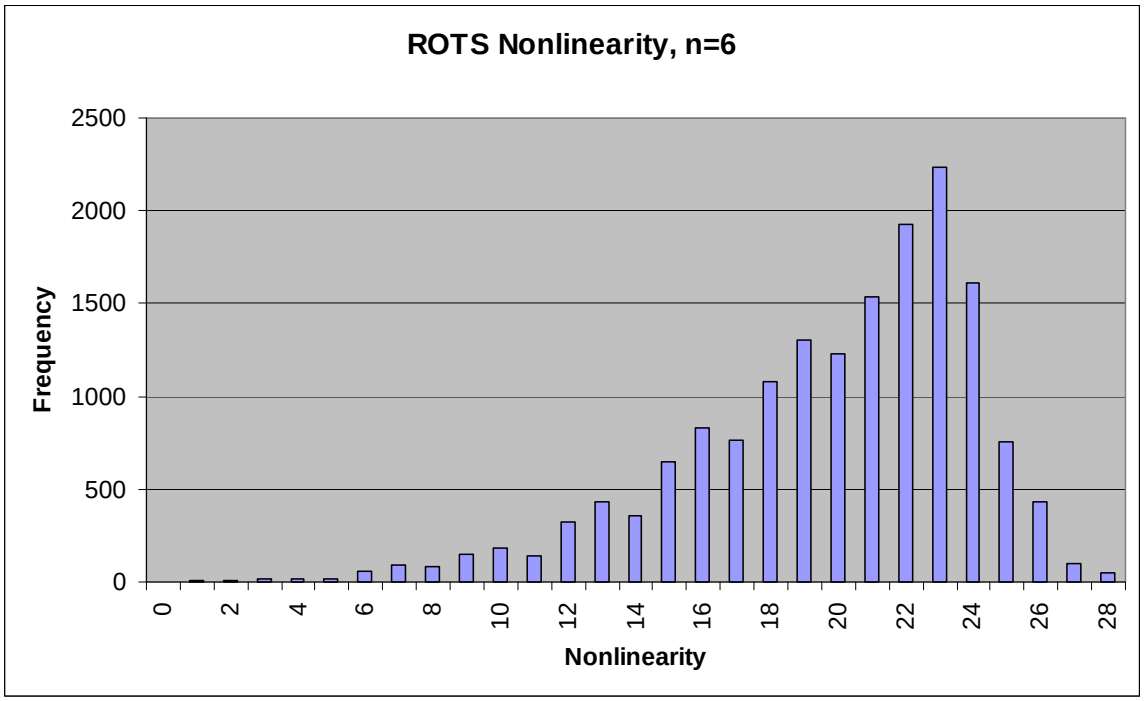

Figure 59 ROTS nonlinearity, $n = 6$

E. ROTS ON $n = 8$

The ability to conduct a nonlinearity calculation on $n = 8$ is hampered by the ever growing size of the circuitry. In order to combat this, two optimization were made to the basic design for the nonlinearity calculation on $n = 6$. The first was implementing the complement optimization described in Chapter II. By doing so, this reduces the number of nonlinearity values going into the minimization circuit by a factor of 2. This, however, does not reduce the complexity of the circuitry to allow a timely compiling. To reduce this complexity by another factor of 2, an additional control signal was provided to the macro from subr.mc. This control signal controlled a two input multiplexer, whose inputs were affine functions. To accommodate for the reduced data throughput in the macro, subr.mc is required to make two calls to the macro.

Consider, for example, the need to count from 0 to 7, in binary. The proper pattern would be 000, 001, 010, … 111. Now suppose you were to count from 0 to 7 again, however, this time you need to count each number twice. This can easily be

accomplished by appending a counting bit to the original three bits. Thus, one is essentially counting from 0 to 15, and using the three most significant bits to indicate the number that is being counted.

A loop is normally used to provide the macro with sequential indices, or sequential blocks of memory containing the functions to be tested. When the loop is providing indices to the macro, it can easily be doubled using the above method. This allows a smaller circuit to be placed on the FPGA, albeit at the expense of a slower execution time. Care must be taken when doing this on three steps. The first is ensuring that the index being sent to the macro in the slowed counter. Using the above example, you would need to be sending the macro 000, 001, … 111 versus 0000, 0001, … 1111. This can be accomplished by conducting a logical right shift by one bit. At the same time, the macro also needs to receive the control signal for the multiplexer in the macro. This can be accomplished by using a logical AND on a counter, and applying that resultant to the macro. Finally, subr.mc needs to compare the nonlinearity value from the most recent call to the macro, with the value from the previous call. The lesser of the two values is chosen and used as the overall nonlinearity for the function under test.

As a result of the enumeration of the 2^{36} ROTS functions on $n = 8$, the following observations were made. 15,104 of the functions are bent, and they are equally divided amongst 3,776 A-classes. That is, there are four ROTS bent functions in each of the 3,776 A-classes. The nonlinearity distribution of the bent functions is shown in Figure 60. Of the total number of ROTS functions, $2.20x10^{-5}$% are bent. The distribution contains all nonlinearities although those below 52 and above 116 are not visible due to their small distribution.

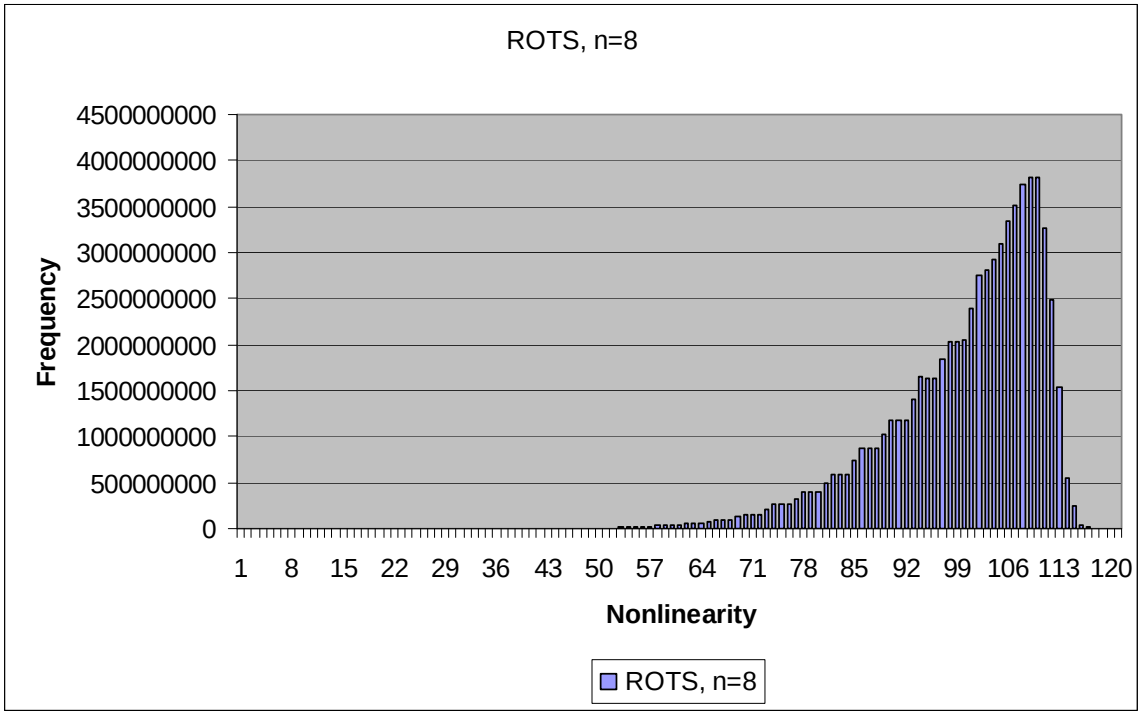

Figure 60 ROTS nonlinearity distribution, $n = 8$

F. SUMMARY

This chapter discussed the various methods used to locate bent functions. It also provides the reader with the necessary knowledge to understand the scarcity of bent functions, and the need to restrict the search space as much as possible.

V. SUMMARY

A. BENT FUNCTIONS

The method best used to find bent functions is based on the number of variables that is being examined. Table 30 shows the scarcity of the bent function. It introduces a new term called the *concentration factor* defined below:

$$cf = \frac{\frac{\#\,bent\ in\ search\ space}{size\ of\ \ search\ space}}{\frac{total\ \#\,bent}{total\ search\ space}}$$

This value shows how changing the size of the search space, e.g. by restricting it to ROTS functions or through searching by degree, the concentration of bent functions becomes noticeably higher. This can become a means to determine which method should be used to locate bent functions for a higher n.

	n=4			n=6			n=8		
	Entire	ROTS	By degree	Entire	ROTS	By degree	Entire	ROTS	By degree
Search space	2^16	2^6	2^6	2^64	2^14	2^35	2^256	2^36	2^154
	65536	64	64	1.84467E+19	16384	34359738368	1.15792E+77	68719476736	2.2836E+46
Total search space	65536	65536	65536	1.84467E+19	1.84467E+19	1.84467E+19	1.15792E+77	1.15792E+77	1.15792E+77
# bent fns	896	8	28	5425430528	48	42386768	9.92706E+31	15104	1.93888E+29
As 2^x	9.807354922	3	4.807354922	32.33709048	5.584962501	25.33711063	106.2911373	13.88264305	97.2911373
fraction bent / search space	0.013671875	0.125	0.4375	2.94113E-10	0.002929688	0.001233617	8.57318E-46	2.19792E-07	8.49046E-18
As 2^x	-6.19264508	-3	-1.19264508	-31.6629095	-8.4150375	-9.66288937	-149.708863	-22.117357	-56.7088627
fraction bent / total search space	0.013671875	0.00012207	0.000427246	2.94113E-10	2.60209E-18	2.29779E-12	8.57318E-46	1.30441E-73	1.67445E-48
As 2^x	-6.19264508	-13	-11.1926451	-31.6629095	-58.4150375	-38.6628894	-149.708863	-242.117357	-158.708863
Concentration factor	1	9.142857143	32	1	9961088.848	4194362.581	1	2.56372E+38	9.90352E+27
As 2^x	0	3.192645078	5	0	23.24787202	22.00002015	0	127.5915058	93

Table 30. Bent function scarcity, After [8]

B. GENETIC ALGORITMS

There is considerable potential in using GAs, and genetic based operations, in order to find bent functions. The most interesting method used a ROTS seed to find bent functions. When using the more traditional approach to GAs, one quarter (6 out of 24) of the available bent functions were located. This is of interest considering that the GA had

processed a total of 77 functions, some of them repeated due to the nature of the GA, that were bent during this time frame. Careful design of the GA, and crosscode selection, will help the GA locate bent functions more rapidly.

C. WHY RECONFIGURABLE COMPUTING

The primary advantage of reconfigurable computing is the ability to work in parallel. Consider the sieving computation for bent functions on 6 variables. The distance between each function and 128 affine functions must be computed. This can be done in parallel. If the GA is applied to a population of 16 chromosomes at one time, this can be considered as a parallel computation involving 16 separate processes acting simultaneously. Despite the relatively few chromosomes in the population for this GA, the reconfigurable computer operating at 100 MHz is quickly capable of outpacing a general purpose computer operating at 2.8 GHz. This can be seen in Figure 61.

The FPGA based GA was translated into C++, and a comparison was conducted. The C++ code was executed on the same CPU that is connected to the FPGA, a 32-bit 2.8 GHz Xeon processor. The C++ implementation includes two versions. The first does not include the half-life and Order 67 logic (C++ simplified), while the second version (C++ full) does. The graph shows the number of CPU clock cycles required to run the GA for a given number of generations. For example, the FPGA GA implementation requires fewer clock cycles compared to C++ full when the number of generations exceeds approximately 7,000. And, it requires fewer clock cycles compared to C++ simplified version when the number of generations exceeds approximately 14,000.

This result can also be seen in Figure 62, which shows the accumulated CPU clock cycles versus the generation. For example, at approximately 7,000 generations and above, the FPGA GA requires fewer clock cycles than C++ full. At approximately 14,000 generations and above, the FPGA requires fewer clock cycles than C++ simplified. The reason that the FPGA curve is relatively flat is because the amount of time it takes to run the GA is insignificant compared to the time required to execute the other instructions. This is similar to the difference in time it takes to list a large amount of files in Linux. For example, it takes considerably less time to list the files while

redirecting the output to a file than to display them to the screen. The number of clock cycles in all cases is determined by using the C "clock()" library function. Since this GA is able to quickly locate fit functions due to the small search space, the FPGA implementation of the GA may be better suited for a larger search space, meaning more bits in the chromosome.

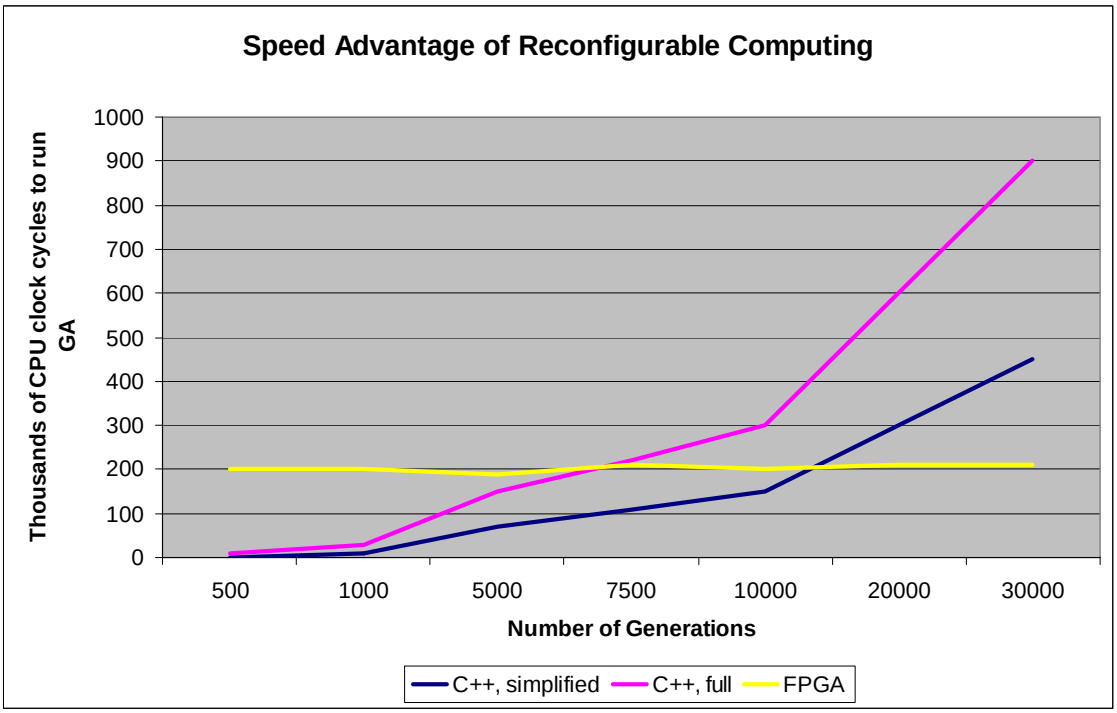

Figure 61 Speed advantage of reconfigurable computing

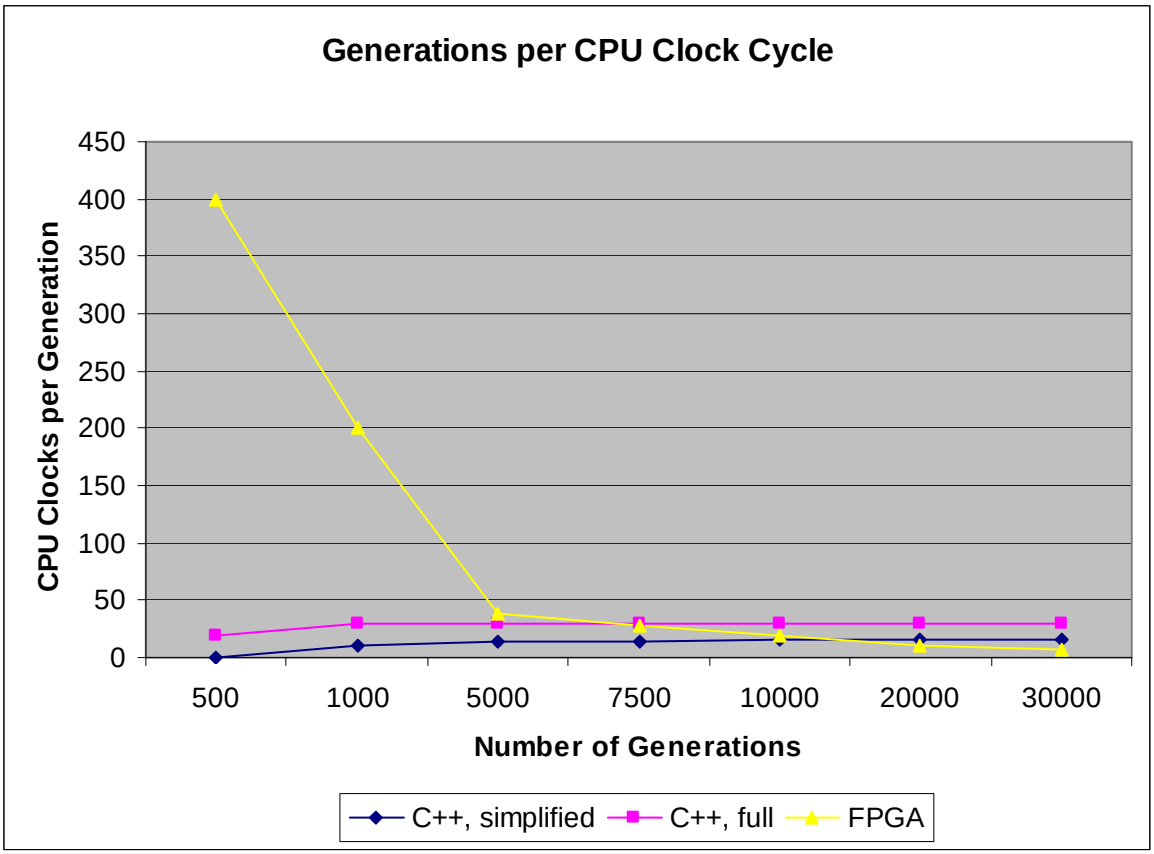

Figure 62 Generations per CPU clock cycle

D. MEETING GOALS

The goal of this thesis was to determine if GAs are useful in finding bent functions. This thesis shows that GAs are a useful tool to locate bent functions. Furthermore, it shows that other genetic processes are useful for generating bent functions from a given bent function. For the first time at NPS we have enumerated all bent functions on $n = 6$ and all ROTS bent functions on $n = 8$. Finally, it was discovered that bent functions can be discovered in groups. This discovery may be useful in a future GA searching for bent functions by degree.

E. FUTURE WORK

Through the course of this thesis several issues were discovered that warrant additional research. This includes the grouping of index adjacent bent functions and

generating bent functions for a ROTS seed. Finally, many sub-programs were created that can be expanded and integrated to facilitate easier testing.

1 ROTS Seed

Additional testing should be done on the ROTS seed method of discovering bent functions. The test should focus on what combinations are required to yield all bent functions. Furthermore, analysis should be conducted on the mathematical properties of the seed functions that yield all bent functions.

Furthermore, this experiment should be expanded to study functions of more variables. It is proposed that the next case to be examined is $n = 8$ since all bent functions on $n = 6$ have already been enumerated in this thesis. The addition of new A-classes of bent functions will provide more research opportunities than revisiting functions of fewer variables.

It is proposed that the ROTS seed algorithm be modified to allow searching for bent functions in more variables. This can be accomplished by programming the co-functions in a hexadecimal character string versus as an integer value. In this method, co-functions can have their bits reversed by simply parsing the character string from the null terminating character to the first character in the string. As each character is parsed, its integer-based hexadecimal value can be computed, and corresponding reversed hexadecimal value be determined through an array based table lookup. Similarly, when two co-functions are exclusive ORed together, this can be done on a character by character basis of their representative hexadecimal strings. Again, the integer based values can be determined from each character. After the two values are exclusive ORed together, the resulting value can be used as an index into a different lookup table to yield the hexadecimal based character of the string. This is all easily accomplished in C++ using the Standard Template Library. Finally, the proposed bent function strings can be printed to a text file which can then already be read by an existing bent function calculator to determine the nonlinearity of the functions.

2. By Bitstuffing on $n = 10$

It is proposed that a ROTS bent function can be created through a similar process to the GA process on $n = 6$. For this case, there are 108 bits in the ROTS index. Of these bins, 99 have a weight of 10. Since the goal is to create a function with 496 ones in it, one of the primary ways to do that is to have 49 of these 99 bins selected, thus giving a weight of 490. The remaining weight can then be evaluated through brute force manipulation of the remaining bins. Building a ROTS index that has exactly 49 ones in it can be done as follows. Create a data structure that has 99 elements in it. Initialize 49 of the elements to a 1, with the remaining to 0. Shuffle the position of the elements through some random process. If this process is to be implemented in C++ with the Standard Template Library, the random_shuffle algorithm can be used to accomplish this. The resulting list can then be converted into a couple of 64 bit numbers by iterating through the list and performing the appropriate bitwise operations on the required variables. This 99 bit value is then processed by the FPGA along with the brute force manipulation of the remaining bits, and the nonlinearity is computed.

In order to perform the nonlinearity calculation on $n = 10$, a method similar to that used on $n = 8$ can be used. In $n = 8$, which affine function that is going to be used is controlled by a control line to a multiplexer. In $n = 10$, which has four times as many bits in the TT, the number of controls lines becomes three. However, instead of a multiplexer being used to store the affine function values, a ROM is instantiated that contains all of the affine functions. It is instantiated 128 times. The value accessed by the ROM is composed of two factors. The first is a constant corresponding to the ROM number. For example, the first ROM would have seven of its address lines being tied to the number 0. Likewise, the last ROM would have seven of its address lines tied to the number 127. The remaining three address lines are provided from by the loop in the subr.mc file. This gives the ROM the ability to access 1,024 different affine functions. As with the case of $n = 8$, a “shortcut” is used that takes into consideration the relationship that an affine function has with its complement on the nonlinearity of a function.

3. Rework on $n = 6$

The fitness function should be rewritten for $n = 6$ to facilitate looking for GA chromosomes that correspond to a ROTS function that has 36 ones instead of 28. Another possibility is to include a version that searches for both conditions, and selects the best case. This can be accomplished by first examining the flowchart created for the current fitness function. The corresponding tables can be recomputed for the new set of circumstances that will result in correctly answering the subproblems.

4. More Efficient Use of Memory Transfers

In order to transfer information between the microprocessor and the FPGAs, a series of memory transfer protocols are used. Currently the algorithm is only using one of the data paths between the two computers. This limits the number of generations that can be run in one execution of the GA. Although this is not a factor in the current GA, the ability to run it over more generations may become necessary in other problems. Furthermore, the memory transfers can be sped up in a few places by utilizing different striping patterns as discussed in the SRC-6 literature.

5. Calculators

Currently a series of calculators are used to send data from one portion of the computation to another. For example, the GA is computed in one executable, and uses input/output redirection to send its data to a text file. The text file is, in turn, read into a different file that converts the chromosome into the normal ROTS index expression, and then into its subsequent truth table. Finally, another program performs the nonlinearity calculations on these truth tables. This cumbersome process is primarily hampered by the need to read text data as a hexadecimal character string and convert it into its numerical value. Furthermore, this is done in groups of eight before the data is passed to the FPGA. This is drastically underutilizing the amount of bandwidth that exists between the microprocessor and the FPGA. The decision to do this is based on the fact that 8 copies of the nonlinearity calculation can be placed on the FPGA at one time. Thus to simplify processing these calculations, the rest of the program was hindered.

These calculators can be rewritten in at least two ways. The first is to send more than 8 values to the FPGA for processing at one time. The most desirable solution would be to combine all of the programs into a single project. In this method, would be recommended for implementation only after all of the calculators are working independently. This is based on the considerable amount of time it takes to compile each of the separate programs. One variant of this would have several subr.mc files in the Makefile, each calling their own calculator. Another version would incorporate all of the calculators into one subr.mc file, and having a control signal from main.c determine which calculator is to be used.

APPENDIX A. STATEFUL MACROS SRC-6

Of interest in this research is the case of a "stateful" macro. As previously mentioned, there are several different possible flags that can be set. One of these flags makes the macro stateful. This is the method that the SRC-6 designers created to implement macros that would retain their state from one iteration of a FOR loop to the next. In order to prevent the macro from continuing to execute when it should not be executing, additional control signals are made to this macro. The different ways to control these signals is beyond the scope of this research. Only their functional result is of interest.

The three control signals that must be implemented for stateful macros are CLEAR, VALID and ITERATION (or ITER). The CLEAR signal would most often be set as a conditional expression in the macro call. An example is when a macro is called for the first time. In this case, the values in the registers should be initialized. Thus, the programmer would control that through the macro call with a conditional expression such as:

my_macro(times==0, in, &out);

In this case, a FOR loop increments the variable times, and when times is 0 (normally during the first execution of the loop), the clear signal is generated and sent to the macro.

The ITER signal is used to ensure that the macro is only processing data once during the iteration of the FOR loop. Various stores to memory might cause the loop to be "slowed down." Loop slow down becomes an issue when several memory writes are required to the same memory bank. When a loop is slowed down, the ITER signal prevents operation of the macro while the FOR loop is processing other lines of code within the FOR loop. The VALID signal only remains high during the time that the macro is being executed.

The final signal, ITER, is used when the number of clocks per iteration of a loop is greater than one. This allows for providing a pulse that is automatically generated once

per iteration. By doing so, an incrementer could be programmed such that it will only increment once per loop iteration. This signal, along with the VALID signal, is useful for creating enable flip-flops.

Figure 62 is taken from [15] to provide a graphical representation of these signals during execution of a stateful macro.

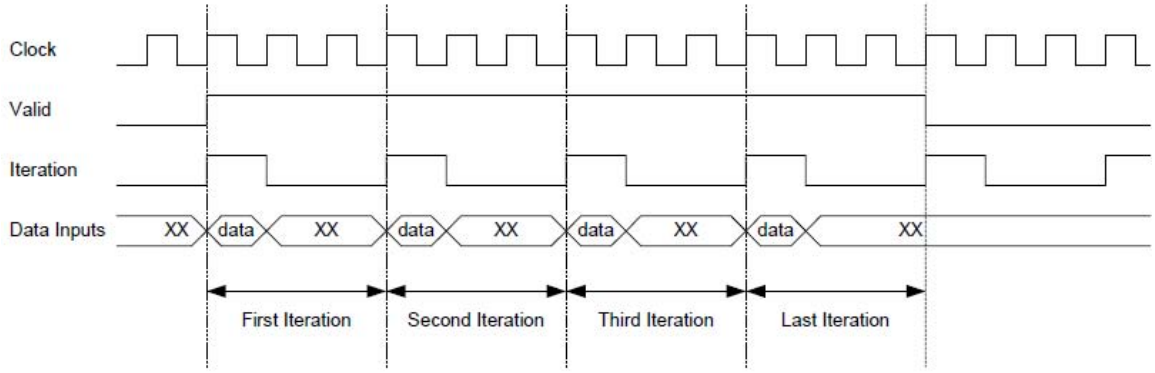

Figure 63 Stateful macro timing diagram, From [15]

APPENDIX B. SRC-6 LESSONS LEARNED

A. MACROS IN A LOOP

When a “function” call is made to a macro, it looks like any other function call made in C. In discussions with Jeff Hammes of SRC Computers, new information was learned about the behavior between the C portion of the code and user defined macros. In a conventional C program, when a function call is made, the appropriate registers are loaded and a jump is executed to the specified portion of memory that contains the function code. This is not the same on the SRC-6 when a function call to a macro is made. Conceptually, each call to a macro is laid down on the FPGA separately. This means that if you have two macro calls, one circuit will be placed on one part of the FPGA, while an identical circuit will be placed at a different part. Now, consider the case where the C code makes a function call from within a FOR loop. Other looping methods can be used to accomplish this. But, for the sake of brevity, only the FOR loop will be discussed. In this case, only one circuit is placed on the FPGA. When the C code is compiled onto the FPGA, the FOR loop is then translated into a machine which sends control signals to the one instance of the macro. This became of interest to this research, since the concept of feedback is necessary for the implementation of the genetic algorithm. It is because of this property that the value in a register stored at the end of iteration of a loop will still be present during the next iteration of the loop. This in effect creates a stateful macro, although without the normal control signals associated with a stateful macro.

B. TIMER ACCESS

The initial idea to provide a random source for ROM address in the macro was to use the timer in the subr.mc file. This is not possible since a call to the timer function made inside a loop will disable loop pipelining. By doing this, the GA macro cannot be made stateful, which the current implementation of the GA requires. Since that is a

considerable undertaking, the decision was made to create random element in the main.c file, and then allow incrimination along with the CRC function to provide the random access to the ROMs.

C. MAKEFILE OPTIONS

An attempt was made to change the mode of operation of the GA in which the latency of the GA was changed to 1, vice the normal 17. In doing so, a new problem would be computed each clock cycle until the feedback brings the old values back to the input. When the normal version of the circuit was implemented, it was always possible to meet the 100.0 MHz timing requirement for the SRC-6. However, changing the latency of the circuit to 1 caused the frequency to drop to 90.8 MHz. After consulting the Xilinx place and route and mapping documentation, options were found that when used sped up the circuit. Table 31 shows the various options that were used and the resulting frequency achieved.

Slowed version of circuit – baseline	100.0 MHz
Fast	90.8 MHz
Utilizing MAP E	92.8 MHz
-timing option	94.0 MHz
Removing “extra” inputs, extra fanouts	100.0 MHz

Table 31. Place and route and mapping options

Normally the microprocessor portion of the program is contained within a file called main.c. However, in order to use some more advanced programming techniques it is desired that C++ be used. In order to accomplish this, two things must be changed. First, the main.c needs to be renamed to main.cc. Second, in the makefile, the linker option needs to be set as follows: LD = icpc.

APPENDIX C. AUXILIARY PROGRAMS

A. INFOER

In order to help produce this project several other side programs were created in C++. The following is a brief description of them. The first is a program called “infoer”. It parses the input Verilog macro and automatically generated the interface files that are required for the C FPGA code to be able to call a user defined macro.

B. CODER

The next program is entitled “coder”. It parses a user input string to produce code. This is helpful in many instances that would normally require repetitive typing. A simple example of this would be in the C portion of the FPGA code. Many times 16 variables need to be passed to and from the Verilog macro. The coder makes it possible to type the following line to produce 16 lines of code that would assign a variable to an element in an array:

!0:15 in%d=IN[%d];\n!

The resultant code that is generated would be:

in0=IN[0]; , in1=IN[1];

and so forth. This has an advantage over the Verilog generate statement. For example, compiling the transeunt triangle for $n > 9$ was not possible since it uses generate statements, and an out of memory error is eventually reached. However, if the transeunt triangle code for $n = 10$ will compile if the generate statements are replaced with the actual lines of code that are to be generated.

C. VERILOG GENERATOR

The next program that was created is called Verilog Generator. It has been expanded from previous projects to include generating the ROMs discussed in this project. This is important because it allows creating new ROMs with either different

words, or more words. It also contains the code necessary to implement the CRC circuitry. Finally, a Verilog standard library was created. This library contains various types of flip-flops, e.g. those with an enable, set, reset, and multiplexors that were used. This proved useful for the times that it is more practical to use structural Verilog versus behavioral.

APPENDIX D. GA CODE

```
module CRC_table_rom(adr, val);
        //Creates a CRC-32 lookup table
        input [7:0] adr;
        output [31:0] val;
        reg [31:0] val;
        always @(adr)
                case (adr)
                0: val = 32'h00000000;
                1: val = 32'h77073096;
                2: val = 32'hee0e612c;
                3: val = 32'h990951ba;
                4: val = 32'h076dc419;
                5: val = 32'h706af48f;
                6: val = 32'he963a535;
                7: val = 32'h9e6495a3;
                8: val = 32'h0edb8832;
                9: val = 32'h79dcb8a4;
                10: val = 32'he0d5e91e;
                11: val = 32'h97d2d988;
                12: val = 32'h09b64c2b;
                13: val = 32'h7eb17cbd;
                14: val = 32'he7b82d07;
                15: val = 32'h90bf1d91;
                16: val = 32'h1db71064;
                17: val = 32'h6ab020f2;
                18: val = 32'hf3b97148;
                19: val = 32'h84be41de;
                20: val = 32'h1adad47d;
                21: val = 32'h6ddde4eb;
                22: val = 32'hf4d4b551;
                23: val = 32'h83d385c7;
                24: val = 32'h136c9856;
                25: val = 32'h646ba8c0;
                26: val = 32'hfd62f97a;
                27: val = 32'h8a65c9ec;
                28: val = 32'h14015c4f;
                29: val = 32'h63066cd9;
                30: val = 32'hfa0f3d63;
                31: val = 32'h8d080df5;
```

```
32: val = 32'h3b6e20c8;
33: val = 32'h4c69105e;
34: val = 32'hd56041e4;
35: val = 32'ha2677172;
36: val = 32'h3c03e4d1;
37: val = 32'h4b04d447;
38: val = 32'hd20d85fd;
39: val = 32'ha50ab56b;
40: val = 32'h35b5a8fa;
41: val = 32'h42b2986c;
42: val = 32'hdbbbc9d6;
43: val = 32'hacbcf940;
44: val = 32'h32d86ce3;
45: val = 32'h45df5c75;
46: val = 32'hdcd60dcf;
47: val = 32'habd13d59;
48: val = 32'h26d930ac;
49: val = 32'h51de003a;
50: val = 32'hc8d75180;
51: val = 32'hbfd06116;
52: val = 32'h21b4f4b5;
53: val = 32'h56b3c423;
54: val = 32'hcfba9599;
55: val = 32'hb8bda50f;
56: val = 32'h2802b89e;
57: val = 32'h5f058808;
58: val = 32'hc60cd9b2;
59: val = 32'hb10be924;
60: val = 32'h2f6f7c87;
61: val = 32'h58684c11;
62: val = 32'hc1611dab;
63: val = 32'hb6662d3d;
64: val = 32'h76dc4190;
65: val = 32'h01db7106;
66: val = 32'h98d220bc;
67: val = 32'hefd5102a;
68: val = 32'h71b18589;
69: val = 32'h06b6b51f;
70: val = 32'h9fbfe4a5;
71: val = 32'he8b8d433;
72: val = 32'h7807c9a2;
73: val = 32'h0f00f934;
```

```
74: val = 32'h9609a88e;
75: val = 32'he10e9818;
76: val = 32'h7f6a0dbb;
77: val = 32'h086d3d2d;
78: val = 32'h91646c97;
79: val = 32'he6635c01;
80: val = 32'h6b6b51f4;
81: val = 32'h1c6c6162;
82: val = 32'h856530d8;
83: val = 32'hf262004e;
84: val = 32'h6c0695ed;
85: val = 32'h1b01a57b;
86: val = 32'h8208f4c1;
87: val = 32'hf50fc457;
88: val = 32'h65b0d9c6;
89: val = 32'h12b7e950;
90: val = 32'h8bbeb8ea;
91: val = 32'hfcb9887c;
92: val = 32'h62dd1ddf;
93: val = 32'h15da2d49;
94: val = 32'h8cd37cf3;
95: val = 32'hfbd44c65;
96: val = 32'h4db26158;
97: val = 32'h3ab551ce;
98: val = 32'ha3bc0074;
99: val = 32'hd4bb30e2;
100: val = 32'h4adfa541;
101: val = 32'h3dd895d7;
102: val = 32'ha4d1c46d;
103: val = 32'hd3d6f4fb;
104: val = 32'h4369e96a;
105: val = 32'h346ed9fc;
106: val = 32'had678846;
107: val = 32'hda60b8d0;
108: val = 32'h44042d73;
109: val = 32'h33031de5;
110: val = 32'haa0a4c5f;
111: val = 32'hdd0d7cc9;
112: val = 32'h5005713c;
113: val = 32'h270241aa;
114: val = 32'hbe0b1010;
115: val = 32'hc90c2086;
```

```
116: val = 32'h5768b525;
117: val = 32'h206f85b3;
118: val = 32'hb966d409;
119: val = 32'hce61e49f;
120: val = 32'h5edef90e;
121: val = 32'h29d9c998;
122: val = 32'hb0d09822;
123: val = 32'hc7d7a8b4;
124: val = 32'h59b33d17;
125: val = 32'h2eb40d81;
126: val = 32'hb7bd5c3b;
127: val = 32'hc0ba6cad;
128: val = 32'hedb88320;
129: val = 32'h9abfb3b6;
130: val = 32'h03b6e20c;
131: val = 32'h74b1d29a;
132: val = 32'head54739;
133: val = 32'h9dd277af;
134: val = 32'h04db2615;
135: val = 32'h73dc1683;
136: val = 32'he3630b12;
137: val = 32'h94643b84;
138: val = 32'h0d6d6a3e;
139: val = 32'h7a6a5aa8;
140: val = 32'he40ecf0b;
141: val = 32'h9309ff9d;
142: val = 32'h0a00ae27;
143: val = 32'h7d079eb1;
144: val = 32'hf00f9344;
145: val = 32'h8708a3d2;
146: val = 32'h1e01f268;
147: val = 32'h6906c2fe;
148: val = 32'hf762575d;
149: val = 32'h806567cb;
150: val = 32'h196c3671;
151: val = 32'h6e6b06e7;
152: val = 32'hfed41b76;
153: val = 32'h89d32be0;
154: val = 32'h10da7a5a;
155: val = 32'h67dd4acc;
156: val = 32'hf9b9df6f;
157: val = 32'h8ebeeff9;
```

```
158: val = 32'h17b7be43;
159: val = 32'h60b08ed5;
160: val = 32'hd6d6a3e8;
161: val = 32'ha1d1937e;
162: val = 32'h38d8c2c4;
163: val = 32'h4fdff252;
164: val = 32'hd1bb67f1;
165: val = 32'ha6bc5767;
166: val = 32'h3fb506dd;
167: val = 32'h48b2364b;
168: val = 32'hd80d2bda;
169: val = 32'haf0a1b4c;
170: val = 32'h36034af6;
171: val = 32'h41047a60;
172: val = 32'hdf60efc3;
173: val = 32'ha867df55;
174: val = 32'h316e8eef;
175: val = 32'h4669be79;
176: val = 32'hcb61b38c;
177: val = 32'hbc66831a;
178: val = 32'h256fd2a0;
179: val = 32'h5268e236;
180: val = 32'hcc0c7795;
181: val = 32'hbb0b4703;
182: val = 32'h220216b9;
183: val = 32'h5505262f;
184: val = 32'hc5ba3bbe;
185: val = 32'hb2bd0b28;
186: val = 32'h2bb45a92;
187: val = 32'h5cb36a04;
188: val = 32'hc2d7ffa7;
189: val = 32'hb5d0cf31;
190: val = 32'h2cd99e8b;
191: val = 32'h5bdeae1d;
192: val = 32'h9b64c2b0;
193: val = 32'hec63f226;
194: val = 32'h756aa39c;
195: val = 32'h026d930a;
196: val = 32'h9c0906a9;
197: val = 32'heb0e363f;
198: val = 32'h72076785;
199: val = 32'h05005713;
```

```
200: val = 32'h95bf4a82;
201: val = 32'he2b87a14;
202: val = 32'h7bb12bae;
203: val = 32'h0cb61b38;
204: val = 32'h92d28e9b;
205: val = 32'he5d5be0d;
206: val = 32'h7cdcefb7;
207: val = 32'h0bdbdf21;
208: val = 32'h86d3d2d4;
209: val = 32'hf1d4e242;
210: val = 32'h68ddb3f8;
211: val = 32'h1fda836e;
212: val = 32'h81be16cd;
213: val = 32'hf6b9265b;
214: val = 32'h6fb077e1;
215: val = 32'h18b74777;
216: val = 32'h88085ae6;
217: val = 32'hff0f6a70;
218: val = 32'h66063bca;
219: val = 32'h11010b5c;
220: val = 32'h8f659eff;
221: val = 32'hf862ae69;
222: val = 32'h616bffd3;
223: val = 32'h166ccf45;
224: val = 32'ha00ae278;
225: val = 32'hd70dd2ee;
226: val = 32'h4e048354;
227: val = 32'h3903b3c2;
228: val = 32'ha7672661;
229: val = 32'hd06016f7;
230: val = 32'h4969474d;
231: val = 32'h3e6e77db;
232: val = 32'haed16a4a;
233: val = 32'hd9d65adc;
234: val = 32'h40df0b66;
235: val = 32'h37d83bf0;
236: val = 32'ha9bcae53;
237: val = 32'hdebb9ec5;
238: val = 32'h47b2cf7f;
239: val = 32'h30b5ffe9;
240: val = 32'hbdbdf21c;
241: val = 32'hcabac28a;
```

```
                242: val = 32'h53b39330;
                243: val = 32'h24b4a3a6;
                244: val = 32'hbad03605;
                245: val = 32'hcdd70693;
                246: val = 32'h54de5729;
                247: val = 32'h23d967bf;
                248: val = 32'hb3667a2e;
                249: val = 32'hc4614ab8;
                250: val = 32'h5d681b02;
                251: val = 32'h2a6f2b94;
                252: val = 32'hb40bbe37;
                253: val = 32'hc30c8ea1;
                254: val = 32'h5a05df1b;
                255: val = 32'h2d02ef8d;
                default: val=32'h0;
                endcase
endmodule

module CRC_calc(in, val);
//module CRC_calc(A, B, C, D, val);
        input [31:0] in;
        //input [7:0] A, B, C, D;
        output [31:0] val;
        wire [31:0] val;
        wire [31:0] regA, regB, regC, regD;
        wire [31:0] topA, topB, topC, topD;

        //CRC_table_rom tA(8'hff^A, topA);
        CRC_table_rom tA(8'hff^in[31:24], topA);
        assign regA= 32'h00ffffff ^ topA;

        //CRC_table_rom tB(regA[7:0] ^ B, topB);
        CRC_table_rom tB(regA[7:0] ^ in[23:16], topB);
        assign regB= {8'h00, regA[31:8]}^ topB;

        //CRC_table_rom tC(regB[7:0] ^ C, topC);
        CRC_table_rom tC(regB[7:0] ^ in[15:8], topC);
        assign regC= {8'h00, regB[31:8]}^ topC;

        //CRC_table_rom tD(regC[7:0] ^ D, topD);
        CRC_table_rom tD(regC[7:0] ^ in[7:0], topD);
        assign regD= {8'h00, regC[31:8]}^ topD;
```

```
		assign val= regD^ 32'hffffffff;
	endmodule


	module OC (TT, Count);
		input[3:0] TT;
		output[2:0] Count;
		wire [2:0] Count;
		assign Count[0]=TT[3]^TT[2]^TT[1]^TT[0];
		assign
Count[1]=(TT[3]&TT[2]|TT[3]&TT[1]|TT[3]&TT[0]|TT[2]&TT[1]|TT[2]&TT[0]|TT[1]&TT[0])&~(TT[3
]&TT[2]&TT[1]&TT[0]);
		assign Count[2]=TT[3]&TT[2]&TT[1]&TT[0];
	endmodule


	module ones64(TT, VALID, RESET, ITER, CLK, count, timer, crcv);
		input [63:0] TT;
		input VALID, RESET, ITER;
		input CLK;
		output [6:0] count;
		reg [6:0] count;
		output [63:0] timer;
		reg [63:0] timer;
		output [31:0] crcv;
		reg [31:0] crcv;


		reg [63:0] d1;
		wire [31:0] w1, w2;


		reg [4:0] counta, countb, countc, countd;
		wire [2:0] count0, count1, count2, count3, count4, count5, count6, count7,
count8, count9, count10, count11, count12, count13, count14, count15;


		CRC_calc cc(TT[31:0], w1);


		dff_eNB ff(w1, VALID&ITER, CLK, w2);


		OC o0(TT[3:0], count0);
		OC o1(TT[7:4], count1);
		OC o2(TT[11:8], count2);
		OC o3(TT[15:12], count3);
		OC o4(TT[19:16], count4);
		OC o5(TT[23:20], count5);
		OC o6(TT[27:24], count6);
```

```
        OC o7(TT[31:28], count7);
        OC o8(TT[35:32], count8);
        OC o9(TT[39:36], count9);
        OC o10(TT[43:40], count10);
        OC o11(TT[47:44], count11);
        OC o12(TT[51:48], count12);
        OC o13(TT[55:52], count13);
        OC o14(TT[59:56], count14);
        OC o15(TT[63:60], count15);

        always @(posedge CLK)
        begin
                counta <=count0+count1+count2+count3;
                countb <=+count4+count5+count6+count7;
                countc <=count8+count9+count10+count11;
                countd <=count12+count13+count14+count15;
                count <=counta+countb+countc+countd;

                d1 <= TT;
                timer <= d1;

                crcv <= w2;
        end
endmodule

module dff_NB(d, CLK, q);
        parameter n=16;
        input [n-1:0] d;
        input CLK;
        output [n-1:0] q;
        reg [n-1:0] q;
        always @(posedge CLK)
                q <= d;
endmodule

module oc5(TT, out);
        input [4:0] TT;
        output [2:0] out;
        reg [2:0] out;

        wire [2:0] val;
        oc4 moc4(TT[3:0], val);
```

```
        always @(TT, val)
                out <= val + TT[4];
    endmodule

    module oc4 (TT, Count);
        input[3:0] TT;
        output[2:0] Count;
        wire [2:0] Count;
        assign Count[0]=TT[3]^TT[2]^TT[1]^TT[0];
        assign
Count[1]=(TT[3]&TT[2]|TT[3]&TT[1]|TT[3]&TT[0]|TT[2]&TT[1]|TT[2]&TT[0]|TT[1]&TT[0])&~(TT[3
]&TT[2]&TT[1]&TT[0]);
        assign Count[2]=TT[3]&TT[2]&TT[1]&TT[0];
    endmodule

    module fit6(TT, CLK, z);
                parameter wid_fit=8;
                parameter wid_TT=14;
         input [wid_TT-1:0] TT;
         input CLK;
         output [wid_fit+wid_TT-1:0] z;
         reg [wid_fit+wid_TT-1:0] z;

         reg [wid_fit-1:0] zd;

         wire [2:0] ones6a, ones6b, ones3, ones2, ones1;
         wire [2:0] ones6ad, ones6bd, ones3d, ones2d, ones1d;
         reg [7:0] ones6;
         reg [7:0] val6, val3, val2, val1;
                reg [wid_TT-1:0] r1_TT;//, r2_TT;
                oc5 moc6a(TT[13:9], ones6ad);
                oc4 moc6b(TT[8:5], ones6bd);
         oc4 moc3({2'b00, TT[4:3]}, ones3d);
         oc4 moc2({3'b000, TT[2]}, ones2d);
         oc4 moc1({2'b0, TT[1:0]}, ones1d);

                defparam f6a.n=3;
                dff_NB f6a(ones6ad, CLK, ones6a);
                defparam f6b.n=3;
                dff_NB f6b(ones6bd, CLK, ones6b);
                defparam f3.n=3;
                dff_NB f3(ones3d, CLK, ones3);
                defparam f2.n=3;
```

```
        dff_NB f2(ones2d, CLK, ones2);
        defparam f1.n=3;
        dff_NB f1(ones1d, CLK, ones1);

        always @(posedge CLK)
        begin
                r1_TT <= TT;
                z[wid_fit+wid_TT-1:wid_fit] <= r1_TT;
                z[wid_fit-1:0] <= zd;
        end
        always @(ones6a, ones6b)
                ones6 <= ones6a + ones6b;


always @(ones6, ones3, ones2, ones1)
begin
        case (ones6)
                3:                      begin
                                                val6 <= 60;
                                                case (ones3)
                                                        0: val3 <=0;
                                                        1: val3 <=30;
                                                        2: val3 <=60;
                                                        default: val3 <=0;
                                                endcase
                                                case (ones2)
                                                        0: val2 <=0;
                                                        1: val2 <=60;
                                                        default: val2 <=0;
                                                endcase
                                                case (ones1)
                                                        0: val1 <=0;
                                                        1: val1 <=30;
                                                        2: val1 <=60;
                                                        default: val1 <=0;
                                                endcase
                                        end
                4:                      begin
                                                val6 <= 80;
                                                if (ones2==1)
                                                begin
                                                        val3 <=0;
```

```
                                                        val2 <=80;
                                                        case (ones1)
                                                                0: val1 <=0;
                                                                1: val1 <=40;
                                                                2: val1 <=80;
                                                                default:   val1
<=0;
                                                        endcase
                                                end
                                                else
                                                begin
                                                        val3 <=80;
                                                        val2 <=0;
                                                        if (ones1==1)
                                                                val1 <=80;
                                                        else
                                                                val1 <=0;
                                                end
                                        end
                                default:        begin
                                                case (ones6)
                                                        0: val6 <=0;
                                                        1: val6 <=10;
                                                        2: val6 <=20;
                                                        5: val6 <=30;
                                                        6: val6 <=40;
                                                        7: val6 <=30;
                                                        8: val6 <=20;
                                                        9: val6 <=10;
                                                        default: val6 <=0;
                                                endcase
                                                case (ones3)
                                                        0: val3 <=15;
                                                        1: val3 <=60;
                                                        2: val3 <=15;
                                                        default: val3 <=0;
                                                endcase
                                                case (ones2)
                                                        0: val2 <=60;
                                                        1: val2 <=15;
                                                        default: val2 <=0;
                                                endcase
```

```
                                                        case (ones1)
                                                                0: val1 <=15;
                                                                1: val1 <=60;
                                                                2: val1 <=15;
                                                                default: val1 <=0;
                                                        endcase
                                                end
                        endcase
                                zd = val6+val3+val2+val1;
                 end
        endmodule



        module dff_cse(d, CLR, SET, EN, CLK, q);
                input d, CLR, SET, EN, CLK;
                output q;
                reg q;
                always @(posedge CLR or posedge SET or posedge CLK)
                        if (CLR)
                                q<=1'b0;
                        else if (SET)
                                q<=1'b1;
                        else if (EN)
                                q<=d;
        endmodule


        module LFSR(CLR, SET, EN, CLK, q);
                parameter n=6;
                //parameter taps=4;
                //Parameter n corresponds to the number of bits in the LFSR
                //Each of the tapX parameters directly corresponds to the maximal tap
                //as shown in Table 3.8 of Dixon

                //In order to properly use the LFSR, the register must first be
initialized with the CLR and SET inputs
                //The value in SET is the first value stored in the register
                //The value of CLR must be the NOT of SET
                //SET and CLR must be LOW after initialization for the LFSR to sequence
```

```
		parameter tap0=0;
		parameter tap1=0;
		parameter tap2=0;
		parameter tap3=0;
		input [n-1:0] CLR, SET;
		input EN, CLK;
		output [n-1:0] q;
		wire [n-1:0] q;
		wire inwire;
		assign inwire = q[tap0]^q[tap1]^q[tap2]^q[tap3];
		dff_cse ff0(inwire, CLR[0], SET[0], EN, CLK, q[0]);
		genvar k;
		generate
		for (k=1; k<n; k=k+1)
			begin: ea_ff
				dff_cse ff(q[k-1], CLR[k], SET[k], EN, CLK, q[k]);
			end
		endgenerate
	endmodule


	module lfsrs(rnd0, rnd1, rnd2, rnd3, rnd4, rnd5, rnd6, rnd7, rnd8, rnd9, rnd10,
rnd11, rnd12, rnd13, rnd14, rnd15, CLR, VALID, ITER, CLK, w_rng0, w_rng1, w_rng2, w_rng3,
w_rng4, w_rng5, w_rng6, w_rng7, w_rng8, w_rng9, w_rng10, w_rng11, w_rng12, w_rng13,
w_rng14, w_rng15);
		//previous is the 14-bit string from the last generation
		//next is the 22-bit string of {14-bit chromosome, 8-bit fitness value}


		parameter n=14;


		parameter AAA=13;
		parameter BBB=4;
		parameter CCC=2;
		parameter DDD=0;


		input [n-1:0] rnd0, rnd1, rnd2, rnd3, rnd4, rnd5, rnd6, rnd7, rnd8, rnd9,
rnd10, rnd11, rnd12, rnd13, rnd14, rnd15;
		input CLR, VALID, ITER, CLK;


		output [n-1:0] w_rng0, w_rng1, w_rng2, w_rng3, w_rng4, w_rng5, w_rng6,
w_rng7, w_rng8, w_rng9, w_rng10, w_rng11, w_rng12, w_rng13, w_rng14, w_rng15;
		wire [n-1:0] w_rng0, w_rng1, w_rng2, w_rng3, w_rng4, w_rng5, w_rng6,
w_rng7, w_rng8, w_rng9, w_rng10, w_rng11, w_rng12, w_rng13, w_rng14, w_rng15;
```

```
wire [n-1:0] clears [15:0];
wire [n-1:0] sets [15:0];


assign sets[0]=rnd0;
assign sets[1]=rnd1;
assign sets[2]=rnd2;
assign sets[3]=rnd3;
assign sets[4]=rnd4;
assign sets[5]=rnd5;
assign sets[6]=rnd6;
assign sets[7]=rnd7;
assign sets[8]=rnd8;
assign sets[9]=rnd9;
assign sets[10]=rnd10;
assign sets[11]=rnd11;
assign sets[12]=rnd12;
assign sets[13]=rnd13;
assign sets[14]=rnd14;
assign sets[15]=rnd15;
assign clears[0]=~sets[0];
assign clears[1]=~sets[1];
assign clears[2]=~sets[2];
assign clears[3]=~sets[3];
assign clears[4]=~sets[4];
assign clears[5]=~sets[5];
assign clears[6]=~sets[6];
assign clears[7]=~sets[7];
assign clears[8]=~sets[8];
assign clears[9]=~sets[9];
assign clears[10]=~sets[10];
assign clears[11]=~sets[11];
assign clears[12]=~sets[12];
assign clears[13]=~sets[13];
assign clears[14]=~sets[14];
assign clears[15]=~sets[15];

defparam rng0.n=14;
defparam rng0.tap0=AAA;
defparam rng0.tap1=BBB;
defparam rng0.tap2=CCC;
defparam rng0.tap3=DDD;
```

```
                LFSR   rng0(clears[0]&{14{CLR}},   sets[0]&{14{CLR}},   VALID&ITER,   CLK,
w_rng0[13:0]);
                defparam rng1.n=14;
                defparam rng1.tap0=AAA;
                defparam rng1.tap1=BBB;
                defparam rng1.tap2=CCC;
                defparam rng1.tap3=DDD;
                LFSR   rng1(clears[1]&{14{CLR}},   sets[1]&{14{CLR}},   VALID&ITER,   CLK,
w_rng1[13:0]);
                defparam rng2.n=14;
                defparam rng2.tap0=AAA;
                defparam rng2.tap1=BBB;
                defparam rng2.tap2=CCC;
                defparam rng2.tap3=DDD;
                LFSR   rng2(clears[2]&{14{CLR}},   sets[2]&{14{CLR}},   VALID&ITER,   CLK,
w_rng2[13:0]);
                defparam rng3.n=14;
                defparam rng3.tap0=AAA;
                defparam rng3.tap1=BBB;
                defparam rng3.tap2=CCC;
                defparam rng3.tap3=DDD;
                LFSR   rng3(clears[3]&{14{CLR}},   sets[3]&{14{CLR}},   VALID&ITER,   CLK,
w_rng3[13:0]);
                defparam rng4.n=14;
                defparam rng4.tap0=AAA;
                defparam rng4.tap1=BBB;
                defparam rng4.tap2=CCC;
                defparam rng4.tap3=DDD;
                LFSR   rng4(clears[4]&{14{CLR}},   sets[4]&{14{CLR}},   VALID&ITER,   CLK,
w_rng4[13:0]);
                defparam rng5.n=14;
                defparam rng5.tap0=AAA;
                defparam rng5.tap1=BBB;
                defparam rng5.tap2=CCC;
                defparam rng5.tap3=DDD;
                LFSR   rng5(clears[5]&{14{CLR}},   sets[5]&{14{CLR}},   VALID&ITER,   CLK,
w_rng5[13:0]);
                defparam rng6.n=14;
                defparam rng6.tap0=AAA;
                defparam rng6.tap1=BBB;
                defparam rng6.tap2=CCC;
                defparam rng6.tap3=DDD;
                LFSR   rng6(clears[6]&{14{CLR}},   sets[6]&{14{CLR}},   VALID&ITER,   CLK,
w_rng6[13:0]);
                defparam rng7.n=14;
```

```
                defparam rng7.tap0=AAA;
                defparam rng7.tap1=BBB;
                defparam rng7.tap2=CCC;
                defparam rng7.tap3=DDD;
                LFSR   rng7(clears[7]&{14{CLR}},   sets[7]&{14{CLR}},   VALID&ITER,   CLK,
w_rng7[13:0]);
                defparam rng8.n=14;
                defparam rng8.tap0=AAA;
                defparam rng8.tap1=BBB;
                defparam rng8.tap2=CCC;
                defparam rng8.tap3=DDD;
                LFSR   rng8(clears[8]&{14{CLR}},   sets[8]&{14{CLR}},   VALID&ITER,   CLK,
w_rng8[13:0]);
                defparam rng9.n=14;
                defparam rng9.tap0=AAA;
                defparam rng9.tap1=BBB;
                defparam rng9.tap2=CCC;
                defparam rng9.tap3=DDD;
                LFSR   rng9(clears[9]&{14{CLR}},   sets[9]&{14{CLR}},   VALID&ITER,   CLK,
w_rng9[13:0]);
                defparam rng10.n=14;
                defparam rng10.tap0=AAA;
                defparam rng10.tap1=BBB;
                defparam rng10.tap2=CCC;
                defparam rng10.tap3=DDD;
                LFSR  rng10(clears[10]&{14{CLR}},  sets[10]&{14{CLR}},  VALID&ITER,  CLK,
w_rng10[13:0]);
                defparam rng11.n=14;
                defparam rng11.tap0=AAA;
                defparam rng11.tap1=BBB;
                defparam rng11.tap2=CCC;
                defparam rng11.tap3=DDD;
                LFSR  rng11(clears[11]&{14{CLR}},  sets[11]&{14{CLR}},  VALID&ITER,  CLK,
w_rng11[13:0]);
                defparam rng12.n=14;
                defparam rng12.tap0=AAA;
                defparam rng12.tap1=BBB;
                defparam rng12.tap2=CCC;
                defparam rng12.tap3=DDD;
                LFSR  rng12(clears[12]&{14{CLR}},  sets[12]&{14{CLR}},  VALID&ITER,  CLK,
w_rng12[13:0]);
                defparam rng13.n=14;
                defparam rng13.tap0=AAA;
                defparam rng13.tap1=BBB;
```

```
		defparam rng13.tap2=CCC;
		defparam rng13.tap3=DDD;
		LFSR  rng13(clears[13]&{14{CLR}},  sets[13]&{14{CLR}},  VALID&ITER,  CLK,
w_rng13[13:0]);
		defparam rng14.n=14;
		defparam rng14.tap0=AAA;
		defparam rng14.tap1=BBB;
		defparam rng14.tap2=CCC;
		defparam rng14.tap3=DDD;
		LFSR  rng14(clears[14]&{14{CLR}},  sets[14]&{14{CLR}},  VALID&ITER,  CLK,
w_rng14[13:0]);
		defparam rng15.n=14;
		defparam rng15.tap0=AAA;
		defparam rng15.tap1=BBB;
		defparam rng15.tap2=CCC;
		defparam rng15.tap3=DDD;
		LFSR  rng15(clears[15]&{14{CLR}},  sets[15]&{14{CLR}},  VALID&ITER,  CLK,
w_rng15[13:0]);

	endmodule


	module slowhalflife(TT, CLR, VALID, ITER, CLK, clear);
		parameter wid_TT=14;
		input [wid_TT-1:0] TT;
		input CLR, VALID, ITER, CLK;
		output clear;
		wire clear;

		reg [wid_TT-1:0] d1, d2, d3;
		reg [2:0] isequal;

		assign clear = &isequal;

		always @(posedge CLK or posedge CLR)
		begin
			if (CLR)
			begin
				d1 <= {wid_TT{1'b1}};
				d2 <= {wid_TT{1'b0}};
				d3 <= {wid_TT{1'b1}};
			end
			else
```

```
                begin
                        if (VALID&ITER)
                        begin
                                d1 <= TT;
                                d2 <= d1;
                                d3 <= d2;
                        end
                end
        end

        always @(TT, d1, d2, d3)
        begin
                isequal[0] <= (TT==d1)?1:0;
                isequal[1] <= (d1==d2)?1:0;
                isequal[2] <= (d2==d3)?1:0;
        end
endmodule


module clearunit(cur0, cur1, cur2, cur3, cur4, cur5, cur6, cur7, cur8, cur9,
cur10, cur11, cur12, cur13, cur14, cur15, CLR, VALID, ITER, CLK, cleared);
        input [13:0] cur0, cur1, cur2, cur3, cur4, cur5, cur6, cur7, cur8, cur9,
cur10, cur11, cur12, cur13, cur14, cur15;
        input CLR, VALID, ITER, CLK;
        output [15:0] cleared;
        wire [15:0] cleared;

        wire [15:0] c1;

        reg [2:0] c2;

        //slowhalflife(TT, CLR, VALID, ITER, CLK, clear);
        slowhalflife st12_0(cur0, CLR, VALID, ITER, CLK, c1[0]);
        slowhalflife st12_1(cur1, CLR, VALID, ITER, CLK, c1[1]);
        slowhalflife st12_2(cur2, CLR, VALID, ITER, CLK, c1[2]);
        slowhalflife st12_3(cur3, CLR, VALID, ITER, CLK, c1[3]);
        slowhalflife st12_4(cur4, CLR, VALID, ITER, CLK, c1[4]);
        slowhalflife st12_5(cur5, CLR, VALID, ITER, CLK, c1[5]);
        slowhalflife st12_6(cur6, CLR, VALID, ITER, CLK, c1[6]);
        slowhalflife st12_7(cur7, CLR, VALID, ITER, CLK, c1[7]);
        slowhalflife st12_8(cur8, CLR, VALID, ITER, CLK, c1[8]);
        slowhalflife st12_9(cur9, CLR, VALID, ITER, CLK, c1[9]);
        slowhalflife st12_10(cur10, CLR, VALID, ITER, CLK, c1[10]);
```

```
		slowhalflife st12_11(cur11, CLR, VALID, ITER, CLK, c1[11]);
		slowhalflife st12_12(cur12, CLR, VALID, ITER, CLK, c1[12]);
		slowhalflife st12_13(cur13, CLR, VALID, ITER, CLK, c1[13]);
		slowhalflife st12_14(cur14, CLR, VALID, ITER, CLK, c1[14]);
		slowhalflife st12_15(cur15, CLR, VALID, ITER, CLK, c1[15]);

		assign cleared[0] = |c2 | CLR | c1[0];
		assign cleared[1] = |c2 | CLR | c1[1];
		assign cleared[2] = |c2 | CLR | c1[2];
		assign cleared[3] = |c2 | CLR | c1[3];
		assign cleared[4] = |c2 | CLR | c1[4];
		assign cleared[5] = |c2 | CLR | c1[5];
		assign cleared[6] = |c2 | CLR | c1[6];
		assign cleared[7] = |c2 | CLR | c1[7];
		assign cleared[8] = |c2 | CLR | c1[8];
		assign cleared[9] = |c2 | CLR | c1[9];
		assign cleared[10] = |c2 | CLR | c1[10];
		assign cleared[11] = |c2 | CLR | c1[11];
		assign cleared[12] = |c2 | CLR | c1[12];
		assign cleared[13] = |c2 | CLR | c1[13];
		assign cleared[14] = |c2 | CLR | c1[14];
		assign cleared[15] = |c2 | CLR | c1[15];

		always @(cur0, cur1, cur2, cur3)
		begin
			c2[0] <= (cur0==cur1)?1:0;
			c2[1] <= (cur1==cur2)?1:0;
			c2[2] <= (cur2==cur3)?1:0;
		end
	endmodule


	module strgen(rnd0, rnd1, rnd2, rnd3, rnd4, rnd5, rnd6, rnd7, rnd8, rnd9, rnd10,
rnd11, rnd12, rnd13, rnd14, rnd15, prev0, prev1, prev2, prev3, prev4, prev5, prev6,
prev7, prev8, prev9, prev10, prev11, prev12, prev13, prev14, prev15, reset_val, CLR,
VALID, ITER, CLK, r_next0, r_next1, r_next2, r_next3, r_next4, r_next5, r_next6, r_next7,
r_next8, r_next9, r_next10,r_next11, r_next12, r_next13, r_next14, r_next15);
		//previous is the 14-bit string from the last generation
		//next is the 22-bit string of {14-bit chromosome, 8-bit fitness value}
		parameter wid_TT=14;
		parameter wid_fit=8;
```

```
		//---------------------------------

		input [13:0] rnd0, rnd1, rnd2, rnd3, rnd4, rnd5, rnd6, rnd7, rnd8, rnd9,
rnd10, rnd11, rnd12, rnd13, rnd14, rnd15;

		input[13:0] prev0, prev1, prev2, prev3, prev4, prev5, prev6, prev7, prev8,
prev9, prev10, prev11, prev12, prev13, prev14, prev15;

		input [7:0] reset_val;


		input CLR, VALID, ITER;

		input CLK /* synthesis syn_noclockbuf=1 syn_maxfan=100000 */ ;


		output [21:0] r_next0, r_next1, r_next2, r_next3, r_next4, r_next5,
r_next6, r_next7, r_next8, r_next9, r_next10,r_next11, r_next12, r_next13, r_next14,
r_next15;

		reg [21:0] r_next0, r_next1, r_next2, r_next3, r_next4, r_next5, r_next6,
r_next7, r_next8, r_next9, r_next10,r_next11, r_next12, r_next13, r_next14, r_next15;


		//---------------------------------


		reg [wid_TT+wid_fit-1:0] next0, next1, next2, next3, next4, next5, next6,
next7, next8, next9, next10, next11, next12, next13, next14, next15;


		reg [13:0] prev0d1, prev1d1, prev2d1, prev3d1, prev4d1, prev5d1, prev6d1,
prev7d1, prev8d1, prev9d1, prev10d1, prev11d1, prev12d1, prev13d1, prev14d1, prev15d1;


		wire [wid_TT-1:0] w_rng0, w_rng1, w_rng2, w_rng3, w_rng4, w_rng5, w_rng6,
w_rng7, w_rng8, w_rng9, w_rng10, w_rng11, w_rng12, w_rng13, w_rng14, w_rng15;




		wire   [wid_TT+wid_fit-1:0]   w_fit_r0,   w_fit_r1,   w_fit_r2,   w_fit_r3,
w_fit_r4,  w_fit_r5,  w_fit_r6,  w_fit_r7,  w_fit_r8,  w_fit_r9,  w_fit_r10,  w_fit_r11,
w_fit_r12, w_fit_r13, w_fit_r14, w_fit_r15;

		wire   [wid_TT+wid_fit-1:0]   w_fit_p0,   w_fit_p1,   w_fit_p2,   w_fit_p3,
w_fit_p4,  w_fit_p5,  w_fit_p6,  w_fit_p7,  w_fit_p8,  w_fit_p9,  w_fit_p10,  w_fit_p11,
w_fit_p12, w_fit_p13, w_fit_p14, w_fit_p15;


		wire [15:0] clearer;


		lfsrs l(rnd0, rnd1, rnd2, rnd3, rnd4, rnd5, rnd6, rnd7, rnd8, rnd9, rnd10,
rnd11, rnd12, rnd13, rnd14, rnd15, CLR, VALID, ITER, CLK, w_rng0, w_rng1, w_rng2, w_rng3,
w_rng4, w_rng5, w_rng6, w_rng7, w_rng8, w_rng9, w_rng10, w_rng11, w_rng12, w_rng13,
w_rng14, w_rng15);


		fit6 fit_r0(w_rng0[13:0], CLK, w_fit_r0);

		fit6 fit_r1(w_rng1[13:0], CLK, w_fit_r1);

		fit6 fit_r2(w_rng2[13:0], CLK, w_fit_r2);

		fit6 fit_r3(w_rng3[13:0], CLK, w_fit_r3);

		fit6 fit_r4(w_rng4[13:0], CLK, w_fit_r4);

		fit6 fit_r5(w_rng5[13:0], CLK, w_fit_r5);
```

```
                fit6 fit_r6(w_rng6[13:0], CLK, w_fit_r6);
                fit6 fit_r7(w_rng7[13:0], CLK, w_fit_r7);
                fit6 fit_r8(w_rng8[13:0], CLK, w_fit_r8);
                fit6 fit_r9(w_rng9[13:0], CLK, w_fit_r9);
                fit6 fit_r10(w_rng10[13:0], CLK, w_fit_r10);
                fit6 fit_r11(w_rng11[13:0], CLK, w_fit_r11);
                fit6 fit_r12(w_rng12[13:0], CLK, w_fit_r12);
                fit6 fit_r13(w_rng13[13:0], CLK, w_fit_r13);
                fit6 fit_r14(w_rng14[13:0], CLK, w_fit_r14);
                fit6 fit_r15(w_rng15[13:0], CLK, w_fit_r15);



                fit6 fit_p0(prev0d1, CLK, w_fit_p0);
                fit6 fit_p1(prev1d1, CLK, w_fit_p1);
                fit6 fit_p2(prev2d1, CLK, w_fit_p2);
                fit6 fit_p3(prev3d1, CLK, w_fit_p3);
                fit6 fit_p4(prev4d1, CLK, w_fit_p4);
                fit6 fit_p5(prev5d1, CLK, w_fit_p5);
                fit6 fit_p6(prev6d1, CLK, w_fit_p6);
                fit6 fit_p7(prev7d1, CLK, w_fit_p7);
                fit6 fit_p8(prev8d1, CLK, w_fit_p8);
                fit6 fit_p9(prev9d1, CLK, w_fit_p9);
                fit6 fit_p10(prev10d1, CLK, w_fit_p10);
                fit6 fit_p11(prev11d1, CLK, w_fit_p11);
                fit6 fit_p12(prev12d1, CLK, w_fit_p12);
                fit6 fit_p13(prev13d1, CLK, w_fit_p13);
                fit6 fit_p14(prev14d1, CLK, w_fit_p14);
                fit6 fit_p15(prev15d1, CLK, w_fit_p15);


                clearunit clru(prev0, prev1, prev2, prev3, prev4, prev5, prev6, prev7,
prev8, prev9, prev10, prev11, prev12, prev13, prev14, prev15, CLR, VALID, ITER, CLK,
clearer);


                always @(*)
                begin
                        next0    [wid_TT+wid_fit-1:0]<=    ((clearer[0])|(w_fit_p0[wid_fit-
1:0]<reset_val[7:0])) ?w_fit_r0:w_fit_p0;
                        next1    [wid_TT+wid_fit-1:0]<=    ((clearer[1])|(w_fit_p1[wid_fit-
1:0]<reset_val[7:0])) ?w_fit_r1:w_fit_p1;
                        next2    [wid_TT+wid_fit-1:0]<=    ((clearer[2])|(w_fit_p2[wid_fit-
1:0]<reset_val[7:0])) ?w_fit_r2:w_fit_p2;
                        next3    [wid_TT+wid_fit-1:0]<=    ((clearer[3])|(w_fit_p3[wid_fit-
1:0]<reset_val[7:0])) ?w_fit_r3:w_fit_p3;
                        next4    [wid_TT+wid_fit-1:0]<=    ((clearer[4])|(w_fit_p4[wid_fit-
1:0]<reset_val[7:0])) ?w_fit_r4:w_fit_p4;
```

```
                        next5    [wid_TT+wid_fit-1:0]<=    ((clearer[5])|(w_fit_p5[wid_fit-
1:0]<reset_val[7:0])) ?w_fit_r5:w_fit_p5;
                        next6    [wid_TT+wid_fit-1:0]<=    ((clearer[6])|(w_fit_p6[wid_fit-
1:0]<reset_val[7:0])) ?w_fit_r6:w_fit_p6;
                        next7    [wid_TT+wid_fit-1:0]<=    ((clearer[7])|(w_fit_p7[wid_fit-
1:0]<reset_val[7:0])) ?w_fit_r7:w_fit_p7;
                        next8    [wid_TT+wid_fit-1:0]<=    ((clearer[8])|(w_fit_p8[wid_fit-
1:0]<reset_val[7:0])) ?w_fit_r8:w_fit_p8;
                        next9    [wid_TT+wid_fit-1:0]<=    ((clearer[9])|(w_fit_p9[wid_fit-
1:0]<reset_val[7:0])) ?w_fit_r9:w_fit_p9;
                        next10   [wid_TT+wid_fit-1:0]<=   ((clearer[10])|(w_fit_p10[wid_fit-
1:0]<reset_val[7:0])) ?w_fit_r10:w_fit_p10;
                        next11   [wid_TT+wid_fit-1:0]<=   ((clearer[11])|(w_fit_p11[wid_fit-
1:0]<reset_val[7:0])) ?w_fit_r11:w_fit_p11;
                        next12   [wid_TT+wid_fit-1:0]<=   ((clearer[12])|(w_fit_p12[wid_fit-
1:0]<reset_val[7:0])) ?w_fit_r12:w_fit_p12;
                        next13   [wid_TT+wid_fit-1:0]<=   ((clearer[13])|(w_fit_p13[wid_fit-
1:0]<reset_val[7:0])) ?w_fit_r13:w_fit_p13;
                        next14   [wid_TT+wid_fit-1:0]<=   ((clearer[14])|(w_fit_p14[wid_fit-
1:0]<reset_val[7:0])) ?w_fit_r14:w_fit_p14;
                        next15   [wid_TT+wid_fit-1:0]<=   ((clearer[15])|(w_fit_p15[wid_fit-
1:0]<reset_val[7:0])) ?w_fit_r15:w_fit_p15;
                end

                always @(posedge CLK)
                begin
                        prev0d1 <= prev0;
                        prev1d1 <= prev1;
                        prev2d1 <= prev2;
                        prev3d1 <= prev3;
                        prev4d1 <= prev4;
                        prev5d1 <= prev5;
                        prev6d1 <= prev6;
                        prev7d1 <= prev7;
                        prev8d1 <= prev8;
                        prev9d1 <= prev9;
                        prev10d1 <= prev10;
                        prev11d1 <= prev11;
                        prev12d1 <= prev12;
                        prev13d1 <= prev13;
                        prev14d1 <= prev14;
                        prev15d1 <= prev15;

                        r_next0 <= next0;
                        r_next1 <= next1;
                        r_next2 <= next2;
```

```
                r_next3 <= next3;
                r_next4 <= next4;
                r_next5 <= next5;
                r_next6 <= next6;
                r_next7 <= next7;
                r_next8 <= next8;
                r_next9 <= next9;
                r_next10 <= next10;
                r_next11 <= next11;
                r_next12 <= next12;
                r_next13 <= next13;
                r_next14 <= next14;
                r_next15 <= next15;

        end
endmodule

module cross_unit(a, b, sel, CLK, w, x, y, z);
        parameter n=14;
        input [n-1:0] a, b, sel;
        input CLK;
        output [n-1:0] w, x, y, z;
        reg [n-1:0] w, x, y, z;
        wire [n-1:0] c, d;
        defparam c1.n=n;
        crossckt c1(a, b, sel, c, d);

        always @(posedge CLK)
        begin
                w <= a;
                x <= b;
                y <= c;
                z <= d;
        end
endmodule



module bit_swap_B(a, b, ctrl, aprime, bprime);
        //If ctrl==1 then swap a and b
        input a, b;
        input ctrl;
```

```
        output aprime, bprime;
        reg aprime, bprime;

        always @(a, b, ctrl)
        begin
                aprime <= ctrl?b:a;
                bprime <= ctrl?a:b;
        end
endmodule


module mux16to1B(A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, sel, Q);
        parameter n=14;
        input [n-1:0] A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P;
        input [3:0] sel;
        output [n-1:0] Q;
        reg [n-1:0] Q;

        always @(A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, sel)
                case (sel)
                        0: Q<=A;
                        1: Q<=B;
                        2: Q<=C;
                        3: Q<=D;
                        4: Q<=E;
                        5: Q<=F;
                        6: Q<=G;
                        7: Q<=H;
                        8: Q<=I;
                        9: Q<=J;
                        10: Q<=K;
                        11: Q<=L;
                        12: Q<=M;
                        13: Q<=N;
                        14: Q<=O;
                        15: Q<=P;
                        default: Q<=A;
                endcase
endmodule

module crossckt(a, b, ctrl, aprime, bprime);
        parameter n=8;
```

```
        input [n-1:0] a, b, ctrl;
        output [n-1:0] aprime, bprime;
        wire [n-1:0] aprime, bprime;

        genvar k;
        generate
                for (k=0; k<n; k=k+1)
                begin: ea_bit
                        bit_swap_B bs(a[k], b[k], ctrl[k], aprime[k], bprime[k]);
                end
        endgenerate
endmodule

module sel_cross(crc_adr, CLK, EN, reset, val);
                input [8:0] crc_adr;
         input CLK, EN, reset;
         output [31:0] val;
         wire [31:0] val;
         reg [7:0] adr;

         reg [7:0] sel_adr;

         crossover_mut_rom machine(sel_adr, val);

         always @(crc_adr, adr)
                sel_adr <= crc_adr[0]?crc_adr[8:1]:adr;

         always @(posedge CLK)
         begin
                 if (reset)
                         adr<=256'd0;
                 else
                                                adr<=adr+EN;
         end
endmodule

module crossover_mut_rom(adr, val);
        //Creates a ROM to control crossover and mutation
        input [7:0] adr;
        output [31:0] val;
        reg [31:0] val;
        always @(adr)
```

```
begin
        case (adr)
                0:      val=32'b00000001001100101010000110010011110;
                1:      val=32'b00100011010001011001110001111011;
                2:      val=32'b01000110000000110001011110001100;
                3:      val=32'b01000110110001010010001110011011;
                4:      val=32'b00110110010000100101100100000111;
                5:      val=32'b11000111000010110101010000110001;
                6:      val=32'b01001110001110010010100001110000;
                7:      val=32'b00000110100001010100111100100011;
                8:      val=32'b00000111000100101010010100110110;
                9:      val=32'b01111010010001010010101100001000;
                10:     val=32'b00101001011000110101111000011011;
                11:     val=32'b11000110000110100100001001011001;
                12:     val=32'b00010111101100100011010111011110;
                13:     val=32'b00100011010101101110000000010111;
                14:     val=32'b01100010001101010000011100010100;
                15:     val=32'b01110100100000010110100100110000;
                16:     val=32'b01010001000011001001001101000111;
                17:     val=32'b00010100001110100010000001100101;
                18:     val=32'b10111010011100100100001110000101;
                19:     val=32'b00010101001001001010001111000110;
                20:     val=32'b11000110100101011000001101000001;
                21:     val=32'b00000010111010101000010001100101;
                22:     val=32'b00101100011010010101010000011010;
                23:     val=32'b01010010110101000000011100110110;
                24:     val=32'b01111011001111000100001001100000;
                25:     val=32'b10011110101101010100110011010010;
                26:     val=32'b00000111000110001110010110111111;
                27:     val=32'b00001001100000110010011100010101;
                28:     val=32'b00001101000101111111100011100011;
                29:     val=32'b11000010111000010011010110100111;
                30:     val=32'b01000110010100011111001000110000;
                31:     val=32'b00110010000001000001101110101001;
                32:     val=32'b01010110110010010000001101110001;
                33:     val=32'b11110000101110000101110110010100;
                34:     val=32'b00111100001000010111010001101011;
                35:     val=32'b10000111000001101001000101000101;
                36:     val=32'b01110110000010000001001010100100;
                37:     val=32'b00011111101101001101001000110101;
                38:     val=32'b01010000000101110011111001100010;
                39:     val=32'b11000101111000010011001010110111;
```

```
40:     val=32'b01001000000110110011100101110101;
41:     val=32'b00011000010100001101011010011010;
42:     val=32'b01110011100010110101010000000010;
43:     val=32'b00101000001101010000101101110100;
44:     val=32'b00010110001101010100101001111100;
45:     val=32'b11110100010100110000101000100110;
46:     val=32'b00100101000011100110010000011100;
47:     val=32'b00110001110110110010010110000000;
48:     val=32'b00011110100101100101001100001010;
49:     val=32'b01010100001111010001000010000010;
50:     val=32'b01000000001010100110010100011011;
51:     val=32'b01101100001010010101000101001000;
52:     val=32'b00000111000110010100101110000010;
53:     val=32'b01010100001100000010000110011100;
54:     val=32'b00100001000010000110101011001001;
55:     val=32'b01100011010111010111000010111100;
56:     val=32'b01010100000000110010100110000111;
57:     val=32'b01110000010000011001001101101101;
58:     val=32'b01001010001000011001010100110110;
59:     val=32'b00000100001011100011100111010001;
60:     val=32'b00100101000011000110011110110011;
61:     val=32'b01110010101001000101001110010000;
62:     val=32'b01110110001000011110001101010000;
63:     val=32'b00101011000110100110001111001110;
64:     val=32'b10110011001010101100000001100001;
65:     val=32'b01110001001001101010010000111001;
66:     val=32'b10100011000000010101011011001001;
67:     val=32'b00000110110001010011011110010100;
68:     val=32'b00000101100001110110001100011001;
69:     val=32'b00100001000010010111100001011111;
70:     val=32'b00011001100001100101101100110010;
71:     val=32'b00110110101001110010010110010001;
72:     val=32'b11100001010000100011100101111011;
73:     val=32'b01100001000010010111111000110101;
74:     val=32'b10000101110101100000001000110111;
75:     val=32'b11000100011000101000000010101110;
76:     val=32'b10010110000010100111001000011000;
77:     val=32'b10010110001101010100000110000111;
78:     val=32'b00001100001100101000010100010111;
79:     val=32'b11100110011100011101101100111010;
80:     val=32'b01011000100101001101000000111010;
81:     val=32'b00000010101100110101110010010001;
```

```
82:     val=32'b10100011000000100110110001010100;
83:     val=32'b00100000000101111100101001101000;
84:     val=32'b01100001010001110011000010001011;
85:     val=32'b11111110010001111000001110010110;
86:     val=32'b01000000010101101000011100010011;
87:     val=32'b11001001101000100000101100010111;
88:     val=32'b01000011101110010000100010100010;
89:     val=32'b01000111010100101000101110100011;
90:     val=32'b10111001011101010011000001000010;
91:     val=32'b01100010000010010011100000010101;
92:     val=32'b10010101000000011101001001100111;
93:     val=32'b10010000000100111000001011111011;
94:     val=32'b01010111101010111100111100111000;
95:     val=32'b00101011010010010011110110100000;
96:     val=32'b10100010111001110011010010000000;
97:     val=32'b11100100110100000011000101011100;
98:     val=32'b00111101011101011000001001001100;
99:     val=32'b00101010010000001001001101010111;
100:    val=32'b00110100111000001000001001101010;
101:    val=32'b01010000001110001001001001000110;
102:    val=32'b01110100111100101010100000011100;
103:    val=32'b10000011000000001011000101010101001;
104:    val=32'b00101100100001001010101110011111;
105:    val=32'b00100100011101101110100000000001;
106:    val=32'b10010101001001000000011010100111;
107:    val=32'b00100000101001000110000110000101;
108:    val=32'b00100000011001110100111100011011;
109:    val=32'b11100001001100100111011010111010;
110:    val=32'b01000110001100000001100101111010;
111:    val=32'b00001111000110101100001101100010;
112:    val=32'b00110100100001010000101001110001;
113:    val=32'b10000001011101011100000010110010;
114:    val=32'b10100011000101010010000011010100;
115:    val=32'b10100101011101100011010000000010;
116:    val=32'b00110100101000101011000000011101;
117:    val=32'b10000011011000000010101110010101;
118:    val=32'b00111100011100101000000000011001;
119:    val=32'b00000110011110010010000100110100;
120:    val=32'b10010100000000010111011010111010;
121:    val=32'b00000001100100110100001011001011;
122:    val=32'b10000000001110100001011101011001;
123:    val=32'b00000001101100110111010000101000;
```

```
124:    val=32'b01010010010010000000011010010001;
125:    val=32'b00111001011101001100010110100010;
126:    val=32'b01100011101000010111110000001000;
127:    val=32'b01100011010100100100011111011010;
128:    val=32'b00010100011001010000001101110010;
129:    val=32'b00000010100100111010100110100110;
130:    val=32'b01010110000111001010101110011101;
131:    val=32'b01110010100100000001010101101000;
132:    val=32'b10000011011011001101001010111001;
133:    val=32'b01111000001101100101000010010100;
134:    val=32'b00000100000100111000001010110110;
135:    val=32'b01011000011110010000011001000010;
136:    val=32'b10100001001001011100011000000011;
137:    val=32'b01011001011100100011010001100001;
138:    val=32'b11000011100000010101001010010100;
139:    val=32'b00100001011110101001010111001110;
140:    val=32'b00100011000001110110101101011100;
141:    val=32'b10111000000101010100011100110010;
142:    val=32'b01000000001110010111010110000001;
143:    val=32'b01010010100101001100111100011000;
144:    val=32'b00000111001101100100000101011100;
145:    val=32'b01101101100000010011010101001001;
146:    val=32'b01011010001101110010000100001101;
147:    val=32'b10000001010000110010100101010000;
148:    val=32'b00010011001001001000000010100101;
149:    val=32'b10010011010000100001010110100111;
150:    val=32'b01100011001000000100010100011010;
151:    val=32'b00010010011101101100111010011010;
152:    val=32'b11010101001000001011010000110001;
153:    val=32'b01010111000010110010111101000110;
154:    val=32'b00001000001101000001011101011100;
155:    val=32'b01001110000000011001101101011010;
156:    val=32'b10101000010001010001000000100110;
157:    val=32'b11010000110001011110000100100110;
158:    val=32'b11010001011111001001011010000101;
159:    val=32'b10010001001110100000001011000100;
160:    val=32'b00100111000111101100011000110101;
161:    val=32'b01100001000001000101110101111000;
162:    val=32'b00100000010111001010010001100111;
163:    val=32'b00011100100101001011111101010010;
164:    val=32'b01110000100001010001101101000011;
165:    val=32'b01110101101101100011001000001010;
```

```
166:    val=32'b00100110000001001000001110110111;
167:    val=32'b01010000101101110100100100110001;
168:    val=32'b00110010000100001011011001001001;
169:    val=32'b00000100100000110010010101101100;
170:    val=32'b00100111001101000110100100001000;
171:    val=32'b00110000000110010101011100101110;
172:    val=32'b00100101000000001101111000011100011;
173:    val=32'b00000010101101000110010011011000l;
174:    val=32'b01001000001000001100001100010101;
175:    val=32'b00101001001101000000011111000101;
176:    val=32'b00010111011000110101010011011010;
177:    val=32'b00100111000100000011011001001000;
178:    val=32'b00100001011101001000010101101010;
179:    val=32'b11000011100110100010010101000000;
180:    val=32'b00010010000010001010011100110100;
181:    val=32'b00000000100101001011001001000010l;
182:    val=32'b00100101000110011100010010000111;
183:    val=32'b00011111100001001001001110100101;
184:    val=32'b11000100100001110010000101100101;
185:    val=32'b10010100000010000011110100100001;
186:    val=32'b10100000100100000011110110101011O;
187:    val=32'b11010100001111111001000001010010l;
188:    val=32'b00010110010010000010011101011110;
189:    val=32'b11010001011110010101001111001000;
190:    val=32'b10001001000000100101000101000011;
191:    val=32'b00011100011010100000001000110100;
192:    val=32'b00011000011001000111000010010010;
193:    val=32'b00010110011101001000100101011010;
194:    val=32'b00111001001000000110011100011010;
195:    val=32'b00011100011100100000011010100100;
196:    val=32'b01110100000101010011011000000010;
197:    val=32'b00010100001110010000101001101000;
198:    val=32'b11000000011010110100100100011010;
199:    val=32'b00111101100001110010010101101110;
200:    val=32'b01000101000110000000001100101011;
201:    val=32'b00000101101010001001001100011110;
202:    val=32'b00101010000111010100011101101001;
203:    val=32'b10100100000010000111001010111111;
204:    val=32'b01100001010110010011000001001101;
205:    val=32'b00011010011110001001011000100011;
206:    val=32'b01010010100001100011110101000001;
207:    val=32'b00010010100101001011001101011111;
```

```
208:    val=32'b00010110010000010110100000010001100;
209:    val=32'b10100110010110001100000000110100;
210:    val=32'b01000000101101111100000111010011;
211:    val=32'b00111010010000000010000111110110;
212:    val=32'b00010111001101000101101011010000;
213:    val=32'b11010110010110010001000001001100;
214:    val=32'b01110100001010100000000110111000;
215:    val=32'b01011011000010010010110000111101;
216:    val=32'b10010000001001010011010000011011;
217:    val=32'b10000011011100100101000000011100;
218:    val=32'b10001101000001000001001001010011;
219:    val=32'b10010100101100011101101010000000;
220:    val=32'b00010100011010100000011110000010;
221:    val=32'b00110101010010000000101101100001;
222:    val=32'b00100001001101100000111010000101;
223:    val=32'b01101011010100000001010001111000;
224:    val=32'b01000001000001011111101001100111;
225:    val=32'b00000111001010011010000101000101;
226:    val=32'b00100111101001000011000110001100;
227:    val=32'b00001001000110100101011000100011;
228:    val=32'b10010001001000110100110101111011;
229:    val=32'b01011010000001100100111100100011;
230:    val=32'b01000101000100001010110001111001;
231:    val=32'b11000001001001111011010100110100;
232:    val=32'b10101001010001100011101101010000;
233:    val=32'b00100001101110001001001101010110;
234:    val=32'b10110010001100001001010001111100;
235:    val=32'b00001011100011010011010001011010;
236:    val=32'b01001011000000100111010111000110;
237:    val=32'b01100000001010100100101101111111;
238:    val=32'b01101010001100010111001010000101;
239:    val=32'b01100000011100110101000101001010;
240:    val=32'b00000011010010000010000101010110;
241:    val=32'b00110101111000101000011100001011;
242:    val=32'b00010011111010110010100000001001000;
243:    val=32'b01110001100101010011101101001000;
244:    val=32'b11000011010110010001101101100100;
245:    val=32'b00000101001010011000101110101101;
246:    val=32'b00111101100100100001101001100101;
247:    val=32'b10100000000100110010010001101001;
248:    val=32'b11010000011001000101101000011001;
249:    val=32'b00100011010110000111000001000001;
```

```
                        250:    val=32'b00010000101000101110001101111111;
                        251:    val=32'b01010011100000001011010001100111;
                        252:    val=32'b10111000000101010010011001111010;
                        253:    val=32'b00010010010000001100001101111001;
                        254:    val=32'b11000011010000001001010101111011;
                        255:    val=32'b00110100100000010010110001010111;
                        default: val=32'd0;
                endcase
        end
endmodule

module dff_cseNB(d, CLR, SET, EN, CLK, q);
        parameter n=32;
        input [n-1:0] d;
        input CLR, SET, EN, CLK;
        output [n-1:0] q;
        reg [n-1:0] q;
        always @(posedge CLR or posedge SET or posedge CLK or posedge EN)
                if (CLR)
                        q<={n{1'b0}};
                else if (SET)
                        q<={n{1'b1}};
                else if (EN)
                        q<=d;
endmodule

module dff_eNB(d, EN, CLK, q);
        parameter n=16;
        input [n-1:0] d;
        input EN, CLK;
        output [n-1:0] q;
        reg [n-1:0] q;
        always @(posedge CLK)
                if (EN)
                        q<=d;
endmodule

module mux2to1B(A, B, sel, Q);
        parameter n=32;
        input [n-1:0] A, B;
        input sel;
        output [n-1:0] Q;
```

```
        reg [n-1:0] Q;

        always @(A, B, sel)
                case (sel)
                        0: Q<=A;
                        1: Q<=B;
                        default: Q<=A;
                endcase
endmodule

module dff_ceNB(d, CLR, EN, CLK, q);
        parameter n=16;
        input [n-1:0] d;
        input CLR, EN, CLK;
        output [n-1:0] q;
        reg [n-1:0] q;
        always @(posedge CLK or posedge CLR)
        begin
                if (CLR)
                        q <= {16{1'b0}};
                else if (EN)
                        q <= d;
        end
endmodule

module MutAdr(inc, CLK, CLR, VALID, ITER, adr);
        parameter n=8;
        input inc, CLK, CLR, VALID, ITER;
        output [7:0] adr;
        wire [7:0] adr;
        wire [7:0] qsum;
        reg [7:0] sum;

        defparam outmux.n=n;
        mux2to1B outmux(sum, {n{1'b0}}, CLR, adr);

        defparam ff.n=n;
        dff_eNB ff(sum, VALID&ITER, CLK, qsum);

        always @(inc or adr)
                sum <= inc + adr;
endmodule
```

```
	module MutRom(ADR, val);
		input [7:0] ADR;
		output [24:0] val;
		reg [24:0] val;
		//Format:  7 bits of 0-run length, wid_TT bits of mutator, 4 bits of which
element selection.

		always @(ADR)
		begin
			case (ADR)
				0:	val=25'b0000010000000001000000011;
				1:	val=25'b0000001000000000000001001;
				2:	val=25'b1111111000001001000001000;
				3:	val=25'b0011101000000000000000010;
				4:	val=25'b0010001000000000000000000;
				5:	val=25'b0010100011000110100100000;
				6:	val=25'b0110010010000001100001100;
				7:	val=25'b1111111000000000000000001;
				8:	val=25'b0011010000000101001001010;
				9:	val=25'b0100010000010000000001010;
				10:	val=25'b0110101001000000001001110;
				11:	val=25'b0000110000000000000011110;
				12:	val=25'b1100011000010000000101111;
				13:	val=25'b1111111000000000000001110;
				14:	val=25'b0001110001101000000000001;
				15:	val=25'b1110011010000100100010001;
				16:	val=25'b0110100010001001000010110;
				17:	val=25'b0100111000010100000001001;
				18:	val=25'b1101011100000000001000011;
				19:	val=25'b1110000100100000000001100;
				20:	val=25'b0111011000001000000101100;
				21:	val=25'b1111111000000000000001101;
				22:	val=25'b0100001000100000100000101;
				23:	val=25'b0000111000000000010001111;
				24:	val=25'b1111111000000000000000011;
				25:	val=25'b1111111000000000000001001;
				26:	val=25'b1010101000000000000001000;
				27:	val=25'b1011000000001000000001010;
				28:	val=25'b1111111000000000000000111;
				29:	val=25'b1111111000000000000001010;
				30:	val=25'b0001100000000000000001100;
```

```
31:     val=25'b011111100010110000000010;
32:     val=25'b111111100000000000000001;
33:     val=25'b011110100001000000001000010;
34:     val=25'b111111100000000000000101;
35:     val=25'b101100010000000000001001000;
36:     val=25'b010000100100001100001000000;
37:     val=25'b001010100000000000000101000;
38:     val=25'b010001010010100000000101101;
39:     val=25'b111111100000000000000000110;
40:     val=25'b010110000001001000000000101;
41:     val=25'b000000100000000000001000000;
42:     val=25'b010011000000000000000000000;
43:     val=25'b100000000000000000000011011;
44:     val=25'b111001010000010000000001111;
45:     val=25'b000000000010000000000001100;
46:     val=25'b001011000000010000000010100;
47:     val=25'b101110100000100000000001001;
48:     val=25'b011111100000000000000001000;
49:     val=25'b011101000001000000000000011;
50:     val=25'b111111100000000000000001001;
51:     val=25'b111111100000000000000001100;
52:     val=25'b000110001000010000000001110;
53:     val=25'b001100000100100010000000001;
54:     val=25'b111101000000000000000000001;
55:     val=25'b001001100000000000000001101;
56:     val=25'b011100000100000000001010101;
57:     val=25'b111011100000000000000001111;
58:     val=25'b101101000000000000000000000;
59:     val=25'b010011100000100000000001000;
60:     val=25'b111111100000000000000001011;
61:     val=25'b111111100000000000000000011;
62:     val=25'b111111100000000000000000110;
63:     val=25'b001110101000010001000010011;
64:     val=25'b111111100000000000000001101;
65:     val=25'b111111100000000000000001011;
66:     val=25'b011100010000010000010101l;
67:     val=25'b000100000000010000000000000;
68:     val=25'b111111100000000000000001001;
69:     val=25'b010011001010010000000000110;
70:     val=25'b010000101000000000000001011;
71:     val=25'b010001100100000000000001011;
72:     val=25'b011100100000010000011010O;
```

```
 73:     val=25'b011000000000010001000000100;
 74:     val=25'b000001100000000000000010010;
 75:     val=25'b111111100000000000000001000;
 76:     val=25'b000010100000001000000011111;
 77:     val=25'b111111100000000000000011111;
 78:     val=25'b010000100100000000000001011;
 79:     val=25'b111111100000000000000011111;
 80:     val=25'b010101000000000000000000100;
 81:     val=25'b001010101000000000000001001;
 82:     val=25'b111111100000000000000000101;
 83:     val=25'b101110100000000110011001001;
 84:     val=25'b000111100000000100100101001;
 85:     val=25'b111111100000000000000000111;
 86:     val=25'b001110100000000000000000101;
 87:     val=25'b111111100000000000000000111;
 88:     val=25'b111111100000000000000000111;
 89:     val=25'b111111100000000000000001110;
 90:     val=25'b010101100000100000011000011;
 91:     val=25'b111001100100000101010000001;
 92:     val=25'b011111000000000000000001011;
 93:     val=25'b111111100000000000000001101;
 94:     val=25'b000101000110000000000101110;
 95:     val=25'b011001000000010000000001100;
 96:     val=25'b100110000000000000000001111;
 97:     val=25'b001100001000000100001101l;
 98:     val=25'b000111100100000000000000010;
 99:     val=25'b000110000010000000001100000;
100:     val=25'b111111100000000000000000100;
101:     val=25'b000101110010010000000000010;
102:     val=25'b001000000000000010000001100;
103:     val=25'b001010000000000010001010l1;
104:     val=25'b111111100000000000000000000;
105:     val=25'b010110000000010001000l1100;
106:     val=25'b000000001000001000000001100;
107:     val=25'b000110100000000000000001011;
108:     val=25'b111101000000000000000000101;
109:     val=25'b011011000000000000000001101;
110:     val=25'b010110110000100000011101l;
111:     val=25'b111111100000000000000001011;
112:     val=25'b111111100000000000000000010;
113:     val=25'b110001100000000010000000l;
114:     val=25'b001101100000000000000001110;
```

```
115:    val=25'b11111110000000000000001010;
116:    val=25'b00111010000000000000001001;
117:    val=25'b01100100000000000010000111;
118:    val=25'b11111110000000000000001011;
119:    val=25'b11111110000000000000000110;
120:    val=25'b11001000000010000010000001;
121:    val=25'b00100100000001000000001110;
122:    val=25'b11010110000000000110001001;
123:    val=25'b10000010000000000000001100;
124:    val=25'b10001001001000001000001110;
125:    val=25'b01010010001001000001101111;
126:    val=25'b11111110000000000000001101;
127:    val=25'b00100010000000000000000000;
128:    val=25'b11111110000000000000001111;
129:    val=25'b00010010000010000000101010;
130:    val=25'b00010110100000000001000011;
131:    val=25'b01011000000000000001000001;
132:    val=25'b00011100100000000000010111;
133:    val=25'b00101000000000010000000101;
134:    val=25'b10100100010000000000001011;
135:    val=25'b10011000000000000000000101;
136:    val=25'b00111010000000010010001111;
137:    val=25'b10010010000000000100000010;
138:    val=25'b00000010000001000000010101;
139:    val=25'b11100000001000001000001100;
140:    val=25'b01001000001000001000001010;
141:    val=25'b00001010010000000000110000;
142:    val=25'b01101110000000000000001111;
143:    val=25'b11111110000000000000000100;
144:    val=25'b11111110000000000000001010;
145:    val=25'b11111110000000000000000101;
146:    val=25'b01100110000000000000001101;
147:    val=25'b11111110000000000000001001;
148:    val=25'b00110110000000001000000010;
149:    val=25'b01111101000000000000001010;
150:    val=25'b10010000000011000000001000;
151:    val=25'b00100100000000000000001111;
152:    val=25'b01111010000000001000101100;
153:    val=25'b00101100000000000010101000;
154:    val=25'b00011010000000000000001011;
155:    val=25'b10111101000001000000000010;
156:    val=25'b00100000100000001000001100;
```

```
157:    val=25'b00111010000000001000000001;
158:    val=25'b01101010000000000100001011;
159:    val=25'b01111001000000000000011110;
160:    val=25'b01111100000001000000000101;
161:    val=25'b00110010000000000001000001;
162:    val=25'b01100000100001001000000110;
163:    val=25'b10110010010000000100001010;
164:    val=25'b11001000000100000010001100;
165:    val=25'b01000100000000111000100110;
166:    val=25'b10100010000000010000001110;
167:    val=25'b10010010000000000000001010;
168:    val=25'b11111110000000000000001010;
169:    val=25'b01111010000100000000001101;
170:    val=25'b11111110000000000000000110;
171:    val=25'b11111110000000000000000101;
172:    val=25'b10001000000000100000000111;
173:    val=25'b01011110100010000000000000;
174:    val=25'b10011000000000100000001101;
175:    val=25'b11111110000000000000000100;
176:    val=25'b11111110000000000000000100;
177:    val=25'b00100010000000000000001101;
178:    val=25'b11111110000000000000001001;
179:    val=25'b00101110000000000000101100;
180:    val=25'b01000010000000000000001000;
181:    val=25'b10001010000000001000010111;
182:    val=25'b00011000000000010000000111;
183:    val=25'b10001000000000000000001101;
184:    val=25'b01011000110010100000001110;
185:    val=25'b00100111000001000010000010;
186:    val=25'b00001000000000100000000001;
187:    val=25'b11001110000000000000001011;
188:    val=25'b10101110000000001100001110;
189:    val=25'b00010010010001000010011011;
190:    val=25'b01000010000000000000001111;
191:    val=25'b11111110000000000000001101;
192:    val=25'b00101000000000011000011001;
193:    val=25'b00011000010000001000100111;
194:    val=25'b11111110000000000000000010;
195:    val=25'b11111110000000000000000101;
196:    val=25'b00010010000000000000001010;
197:    val=25'b00000100000100000010011101;
198:    val=25'b11101100000000001000001110;
```

```
199:    val=25'b10001110010010000000001111;
200:    val=25'b11111110000000000000001010;
201:    val=25'b01001000000000000000010110;
202:    val=25'b11111110000000000000000110;
203:    val=25'b00001110000010000000101101;
204:    val=25'b01111100000000000000000011;
205:    val=25'b00100100000000000010000011;
206:    val=25'b10011000000000000000001101;
207:    val=25'b01100010000000000000010001;
208:    val=25'b00100011000000010100001000;
209:    val=25'b01000000000011000000001111;
210:    val=25'b01110110000000000000011011;
211:    val=25'b11111110000000000000001100;
212:    val=25'b01010001000000000000100011;
213:    val=25'b11100010010000000000000111;
214:    val=25'b01110110000000000000001110;
215:    val=25'b11111110000000000000000110;
216:    val=25'b01101010000010000000000000;
217:    val=25'b01001110000000000001000111;
218:    val=25'b00000100000000001000000010;
219:    val=25'b00010100000000000000001111;
220:    val=25'b00000100001000000100001010;
221:    val=25'b11111110000000000000000110;
222:    val=25'b11100100000000000000000010;
223:    val=25'b11111110000000000000000110;
224:    val=25'b11111110000000000000000011;
225:    val=25'b11111110000000000000000100;
226:    val=25'b11111110000000000000000000;
227:    val=25'b00110010000000000000001000;
228:    val=25'b11111110000000000000001000;
229:    val=25'b01011000000000000000000011;
230:    val=25'b10011110000000010100000010;
231:    val=25'b01111000000000000000001011;
232:    val=25'b11111000000000000000000101;
233:    val=25'b00010000000000000010000100;
234:    val=25'b11011110000000010000001110;
235:    val=25'b01010100000000000000001011;
236:    val=25'b00100010001000010000000101;
237:    val=25'b00111111000000000000000011;
238:    val=25'b00110000000000100010001000;
239:    val=25'b00100100010010000000000100;
240:    val=25'b10010101000000000000001110;
```

```
                        241:    val=25'b10011010001000000000000101;
                        242:    val=25'b00111010000000000000001011;
                        243:    val=25'b00111000001000000000000001;
                        244:    val=25'b11111110000000000000001100;
                        245:    val=25'b10010110000000000000000001;
                        246:    val=25'b11111110000000000000000101;
                        247:    val=25'b01000000000000000000000100;
                        248:    val=25'b00000000000001000000001011;
                        249:    val=25'b00111000001100000001001100;
                        250:    val=25'b10100100100000000001000111;
                        251:    val=25'b11111110000000000000000110;
                        252:    val=25'b01011110000000000000001001;
                        253:    val=25'b00001100100110000000011001;
                        254:    val=25'b01000110000000000000101011;
                        255:    val=25'b11101110000000000000000001;
                        default: val=25'd0;
                endcase
        end
endmodule

module incer(val, CLR, EN, CLK, out);
        parameter n=8;
        input val, CLR, EN, CLK;
        output [n-1:0] out;
        reg [n-1:0] out;
        always @(posedge CLK or posedge CLR)
                if (CLR)
                        out <= {n{1'b0}};
                else if (EN)
                        out <= out + val;
endmodule

module muter(crc_adr, CLR, VALID, ITER, CLK, r_mutout);
        parameter wrom=25;   //number of bits in ROM lines, equals {7'Zeros run,
14'Mutation code, 4'element selection}
        parameter alines=8; //number of bits in ROM address line
        parameter maxrun=7; //number of bits in countre

        input [8:0] crc_adr;
        input CLR, VALID, ITER, CLK;
        output [17:0] r_mutout;
        reg [17:0] r_mutout;
```

```
		wire [17:0] mutout;

		wire [wrom-1:0] romout;
		wire w_isnequal;
		wire [alines-1:0] wq_adr;
		wire [maxrun-1:0] wq_ctr;
		reg r_isequal, d_isequal;

		reg [7:0] rom_adr;

		assign w_isnequal = ~r_isequal;

		defparam inc_ctr.n=maxrun;

		incer inc_ctr(~r_isequal, d_isequal|CLR, VALID&ITER, CLK, wq_ctr);

		always @(wq_ctr, romout[wrom-1:wrom-maxrun])
			r_isequal <= wq_ctr == romout[wrom-1:wrom-maxrun];

		always @(posedge CLK)// or posedge CLR)
		begin
				d_isequal <= r_isequal;
				r_mutout <= mutout;
		end

		defparam inc_adr.n=alines;
		incer inc_adr(d_isequal, CLR, VALID&ITER, CLK, wq_adr);

		always @(crc_adr, wq_adr)
			rom_adr <= crc_adr[0]?crc_adr[8:1]:wq_adr;

		MutRom mr(rom_adr, romout);

		defparam m_mutout.n=wrom-maxrun;
		mux2to1B	m_mutout({(wrom-maxrun){1'b0}},	romout[(wrom-maxrun-1):0],
r_isequal, mutout);


	endmodule


	module crossmut(w_sort0, w_sort1, w_sort2, w_sort3, w_sort4, w_sort5, w_sort6,
w_sort7, w_sort8, w_sort9, w_sort10, w_sort11, w_sort12, w_sort13, w_sort14, w_sort15,
crc_adr, crosscode, CLR, VALID, ITER, CLK, r_mout0, r_mout1, r_mout2, r_mout3, r_mout4,
```

```
r_mout5, r_mout6, r_mout7, r_mout8, r_mout9, r_mout10, r_mout11, r_mout12, r_mout13,
r_mout14, r_mout15);

        parameter wid_TT=14;

        parameter wid_fit=8;


        input [21:0] w_sort0, w_sort1, w_sort2, w_sort3, w_sort4, w_sort5,
w_sort6, w_sort7, w_sort8, w_sort9, w_sort10, w_sort11, w_sort12, w_sort13, w_sort14,
w_sort15;


        input [8:0] crc_adr;

        input [13:0] crosscode;

        //crc adr has the format of 8:1 adr lines, 0 control line

        input CLR;

        input VALID, ITER;

        input CLK /* synthesis syn_noclockbuf=1 syn_maxfan=100000 */;

        output [13:0] r_mout0, r_mout1, r_mout2, r_mout3, r_mout4, r_mout5,
r_mout6, r_mout7, r_mout8, r_mout9, r_mout10, r_mout11, r_mout12, r_mout13, r_mout14,
r_mout15;

        reg [13:0] r_mout0, r_mout1, r_mout2, r_mout3, r_mout4, r_mout5, r_mout6,
r_mout7, r_mout8, r_mout9, r_mout10, r_mout11, r_mout12, r_mout13, r_mout14, r_mout15;

        wire [13:0] w_mout0, w_mout1, w_mout2, w_mout3, w_mout4, w_mout5, w_mout6,
w_mout7, w_mout8, w_mout9, w_mout10, w_mout11, w_mout12, w_mout13, w_mout14, w_mout15;

        wire [13:0] w_out0, w_out1, w_out2, w_out3, w_out4, w_out5, w_out6,
w_out7, w_out8, w_out9, w_out10, w_out11, w_out12, w_out13, w_out14, w_out15;


        wire [31:0] sel;


        wire [wid_TT-1:0] w_m0, w_m1, w_m2, w_m3, w_m4, w_m5, w_m6, w_m7;

        reg [wid_TT-1:0] r_m0, r_m1, r_m2, r_m3, r_m4, r_m5, r_m6, r_m7;


        reg w_selmut0, w_selmut1, w_selmut2, w_selmut3, w_selmut4, w_selmut5,
w_selmut6, w_selmut7, w_selmut8, w_selmut9, w_selmut10, w_selmut11, w_selmut12,
w_selmut13, w_selmut14, w_selmut15;

        reg [13:0] w_muted0, w_muted1, w_muted2, w_muted3, w_muted4, w_muted5,
w_muted6, w_muted7, w_muted8, w_muted9, w_muted10, w_muted11, w_muted12, w_muted13,
w_muted14, w_muted15;


        //sel_cross(CLK, EN, reset, val);

        sel_cross selx(crc_adr, CLK, VALID&ITER, CLR, sel);



        //wire [wid_TT-1:0] crosscode;

        //assign crosscode = 14'b11111111100000;

        //assign crosscode = 14'b00000011111111;


        wire [17:0] w_mutout;

        //muter(CLR, VALID, ITER, CLK, r_mutout);
```

```
            muter mtr(crc_adr, CLR, VALID, ITER, CLK, w_mutout);


            defparam m0.n=wid_TT;

            defparam m1.n=wid_TT;

            defparam m2.n=wid_TT;

            defparam m3.n=wid_TT;

            defparam m4.n=wid_TT;

            defparam m5.n=wid_TT;

            defparam m6.n=wid_TT;

            defparam m7.n=wid_TT;

            mux16to1B   m0(w_sort0[wid_TT+wid_fit-1:wid_fit],   w_sort1[wid_TT+wid_fit-
1:wid_fit],     w_sort2[wid_TT+wid_fit-1:wid_fit],     w_sort3[wid_TT+wid_fit-1:wid_fit],
w_sort4[wid_TT+wid_fit-1:wid_fit],                     w_sort5[wid_TT+wid_fit-1:wid_fit],
w_sort6[wid_TT+wid_fit-1:wid_fit],                     w_sort7[wid_TT+wid_fit-1:wid_fit],
w_sort8[wid_TT+wid_fit-1:wid_fit],                     w_sort9[wid_TT+wid_fit-1:wid_fit],
w_sort10[wid_TT+wid_fit-1:wid_fit],                   w_sort11[wid_TT+wid_fit-1:wid_fit],
w_sort12[wid_TT+wid_fit-1:wid_fit],                   w_sort13[wid_TT+wid_fit-1:wid_fit],
w_sort14[wid_TT+wid_fit-1:wid_fit],    w_sort15[wid_TT+wid_fit-1:wid_fit],    sel[31:28],
w_m0);

            mux16to1B   m1(w_sort0[wid_TT+wid_fit-1:wid_fit],   w_sort1[wid_TT+wid_fit-
1:wid_fit],     w_sort2[wid_TT+wid_fit-1:wid_fit],     w_sort3[wid_TT+wid_fit-1:wid_fit],
w_sort4[wid_TT+wid_fit-1:wid_fit],                     w_sort5[wid_TT+wid_fit-1:wid_fit],
w_sort6[wid_TT+wid_fit-1:wid_fit],                     w_sort7[wid_TT+wid_fit-1:wid_fit],
w_sort8[wid_TT+wid_fit-1:wid_fit],                     w_sort9[wid_TT+wid_fit-1:wid_fit],
w_sort10[wid_TT+wid_fit-1:wid_fit],                   w_sort11[wid_TT+wid_fit-1:wid_fit],
w_sort12[wid_TT+wid_fit-1:wid_fit],                   w_sort13[wid_TT+wid_fit-1:wid_fit],
w_sort14[wid_TT+wid_fit-1:wid_fit],    w_sort15[wid_TT+wid_fit-1:wid_fit],    sel[27:24],
w_m1);

            mux16to1B   m2(w_sort0[wid_TT+wid_fit-1:wid_fit],   w_sort1[wid_TT+wid_fit-
1:wid_fit],     w_sort2[wid_TT+wid_fit-1:wid_fit],     w_sort3[wid_TT+wid_fit-1:wid_fit],
w_sort4[wid_TT+wid_fit-1:wid_fit],                     w_sort5[wid_TT+wid_fit-1:wid_fit],
w_sort6[wid_TT+wid_fit-1:wid_fit],                     w_sort7[wid_TT+wid_fit-1:wid_fit],
w_sort8[wid_TT+wid_fit-1:wid_fit],                     w_sort9[wid_TT+wid_fit-1:wid_fit],
w_sort10[wid_TT+wid_fit-1:wid_fit],                   w_sort11[wid_TT+wid_fit-1:wid_fit],
w_sort12[wid_TT+wid_fit-1:wid_fit],                   w_sort13[wid_TT+wid_fit-1:wid_fit],
w_sort14[wid_TT+wid_fit-1:wid_fit],    w_sort15[wid_TT+wid_fit-1:wid_fit],    sel[23:20],
w_m2);

            mux16to1B   m3(w_sort0[wid_TT+wid_fit-1:wid_fit],   w_sort1[wid_TT+wid_fit-
1:wid_fit],     w_sort2[wid_TT+wid_fit-1:wid_fit],     w_sort3[wid_TT+wid_fit-1:wid_fit],
w_sort4[wid_TT+wid_fit-1:wid_fit],                     w_sort5[wid_TT+wid_fit-1:wid_fit],
w_sort6[wid_TT+wid_fit-1:wid_fit],                     w_sort7[wid_TT+wid_fit-1:wid_fit],
w_sort8[wid_TT+wid_fit-1:wid_fit],                     w_sort9[wid_TT+wid_fit-1:wid_fit],
w_sort10[wid_TT+wid_fit-1:wid_fit],                   w_sort11[wid_TT+wid_fit-1:wid_fit],
w_sort12[wid_TT+wid_fit-1:wid_fit],                   w_sort13[wid_TT+wid_fit-1:wid_fit],
w_sort14[wid_TT+wid_fit-1:wid_fit],    w_sort15[wid_TT+wid_fit-1:wid_fit],    sel[19:16],
w_m3);

            mux16to1B   m4(w_sort0[wid_TT+wid_fit-1:wid_fit],   w_sort1[wid_TT+wid_fit-
1:wid_fit],     w_sort2[wid_TT+wid_fit-1:wid_fit],     w_sort3[wid_TT+wid_fit-1:wid_fit],
w_sort4[wid_TT+wid_fit-1:wid_fit],                     w_sort5[wid_TT+wid_fit-1:wid_fit],
w_sort6[wid_TT+wid_fit-1:wid_fit],                     w_sort7[wid_TT+wid_fit-1:wid_fit],
w_sort8[wid_TT+wid_fit-1:wid_fit],                     w_sort9[wid_TT+wid_fit-1:wid_fit],
w_sort10[wid_TT+wid_fit-1:wid_fit],                   w_sort11[wid_TT+wid_fit-1:wid_fit],
w_sort12[wid_TT+wid_fit-1:wid_fit],                   w_sort13[wid_TT+wid_fit-1:wid_fit],
w_sort14[wid_TT+wid_fit-1:wid_fit],    w_sort15[wid_TT+wid_fit-1:wid_fit],    sel[15:12],
w_m4);

            mux16to1B   m5(w_sort0[wid_TT+wid_fit-1:wid_fit],   w_sort1[wid_TT+wid_fit-
1:wid_fit],     w_sort2[wid_TT+wid_fit-1:wid_fit],     w_sort3[wid_TT+wid_fit-1:wid_fit],
w_sort4[wid_TT+wid_fit-1:wid_fit],                     w_sort5[wid_TT+wid_fit-1:wid_fit],
w_sort6[wid_TT+wid_fit-1:wid_fit],                     w_sort7[wid_TT+wid_fit-1:wid_fit],
w_sort8[wid_TT+wid_fit-1:wid_fit],                     w_sort9[wid_TT+wid_fit-1:wid_fit],
```

```
w_sort10[wid_TT+wid_fit-1:wid_fit],                    w_sort11[wid_TT+wid_fit-1:wid_fit],
w_sort12[wid_TT+wid_fit-1:wid_fit],                    w_sort13[wid_TT+wid_fit-1:wid_fit],
w_sort14[wid_TT+wid_fit-1:wid_fit], w_sort15[wid_TT+wid_fit-1:wid_fit], sel[11:8], w_m5);

              mux16to1B   m6(w_sort0[wid_TT+wid_fit-1:wid_fit],   w_sort1[wid_TT+wid_fit-
1:wid_fit],     w_sort2[wid_TT+wid_fit-1:wid_fit],     w_sort3[wid_TT+wid_fit-1:wid_fit],
w_sort4[wid_TT+wid_fit-1:wid_fit],                     w_sort5[wid_TT+wid_fit-1:wid_fit],
w_sort6[wid_TT+wid_fit-1:wid_fit],                     w_sort7[wid_TT+wid_fit-1:wid_fit],
w_sort8[wid_TT+wid_fit-1:wid_fit],                     w_sort9[wid_TT+wid_fit-1:wid_fit],
w_sort10[wid_TT+wid_fit-1:wid_fit],                    w_sort11[wid_TT+wid_fit-1:wid_fit],
w_sort12[wid_TT+wid_fit-1:wid_fit],                    w_sort13[wid_TT+wid_fit-1:wid_fit],
w_sort14[wid_TT+wid_fit-1:wid_fit], w_sort15[wid_TT+wid_fit-1:wid_fit], sel[7:4], w_m6);

              mux16to1B   m7(w_sort0[wid_TT+wid_fit-1:wid_fit],   w_sort1[wid_TT+wid_fit-
1:wid_fit],     w_sort2[wid_TT+wid_fit-1:wid_fit],     w_sort3[wid_TT+wid_fit-1:wid_fit],
w_sort4[wid_TT+wid_fit-1:wid_fit],                     w_sort5[wid_TT+wid_fit-1:wid_fit],
w_sort6[wid_TT+wid_fit-1:wid_fit],                     w_sort7[wid_TT+wid_fit-1:wid_fit],
w_sort8[wid_TT+wid_fit-1:wid_fit],                     w_sort9[wid_TT+wid_fit-1:wid_fit],
w_sort10[wid_TT+wid_fit-1:wid_fit],                    w_sort11[wid_TT+wid_fit-1:wid_fit],
w_sort12[wid_TT+wid_fit-1:wid_fit],                    w_sort13[wid_TT+wid_fit-1:wid_fit],
w_sort14[wid_TT+wid_fit-1:wid_fit], w_sort15[wid_TT+wid_fit-1:wid_fit], sel[3:0], w_m7);


              defparam mutmux0.n=wid_TT;

              defparam mutmux1.n=wid_TT;

              defparam mutmux2.n=wid_TT;

              defparam mutmux3.n=wid_TT;

              defparam mutmux4.n=wid_TT;

              defparam mutmux5.n=wid_TT;

              defparam mutmux6.n=wid_TT;

              defparam mutmux7.n=wid_TT;

              defparam mutmux8.n=wid_TT;

              defparam mutmux9.n=wid_TT;

              defparam mutmux10.n=wid_TT;

              defparam mutmux11.n=wid_TT;

              defparam mutmux12.n=wid_TT;

              defparam mutmux13.n=wid_TT;

              defparam mutmux14.n=wid_TT;

              defparam mutmux15.n=wid_TT;

              mux2to1B mutmux0(w_out0, w_muted0, w_selmut0, w_mout0);

              mux2to1B mutmux1(w_out1, w_muted1, w_selmut1, w_mout1);

              mux2to1B mutmux2(w_out2, w_muted2, w_selmut2, w_mout2);

              mux2to1B mutmux3(w_out3, w_muted3, w_selmut3, w_mout3);

              mux2to1B mutmux4(w_out4, w_muted4, w_selmut4, w_mout4);

              mux2to1B mutmux5(w_out5, w_muted5, w_selmut5, w_mout5);

              mux2to1B mutmux6(w_out6, w_muted6, w_selmut6, w_mout6);

              mux2to1B mutmux7(w_out7, w_muted7, w_selmut7, w_mout7);

              mux2to1B mutmux8(w_out8, w_muted8, w_selmut8, w_mout8);

              mux2to1B mutmux9(w_out9, w_muted9, w_selmut9, w_mout9);

              mux2to1B mutmux10(w_out10, w_muted10, w_selmut10, w_mout10);

              mux2to1B mutmux11(w_out11, w_muted11, w_selmut11, w_mout11);
```

```
mux2to1B mutmux12(w_out12, w_muted12, w_selmut12, w_mout12);
mux2to1B mutmux13(w_out13, w_muted13, w_selmut13, w_mout13);
mux2to1B mutmux14(w_out14, w_muted14, w_selmut14, w_mout14);
mux2to1B mutmux15(w_out15, w_muted15, w_selmut15, w_mout15);

defparam cu0.n=wid_TT;
cross_unit cu0(r_m0, r_m1, crosscode, CLK, w_out0, w_out1, w_out2,
w_out3);

defparam cu1.n=wid_TT;
cross_unit cu1(r_m2, r_m3, crosscode, CLK, w_out4, w_out5, w_out6,
w_out7);

defparam cu2.n=wid_TT;
cross_unit cu2(r_m4, r_m5, crosscode, CLK, w_out8, w_out9, w_out10,
w_out11);

defparam cu3.n=wid_TT;
cross_unit cu3(r_m6, r_m7, crosscode, CLK, w_out12, w_out13, w_out14,
w_out15);

always @(posedge CLK)
begin
        r_m0 <= w_m0;
        r_m1 <= w_m1;
        r_m2 <= w_m2;
        r_m3 <= w_m3;
        r_m4 <= w_m4;
        r_m5 <= w_m5;
        r_m6 <= w_m6;
        r_m7 <= w_m7;
        r_mout0 <= w_mout0;
        r_mout1 <= w_mout1;
        r_mout2 <= w_mout2;
        r_mout3 <= w_mout3;
        r_mout4 <= w_mout4;
        r_mout5 <= w_mout5;
        r_mout6 <= w_mout6;
        r_mout7 <= w_mout7;
        r_mout8 <= w_mout8;
        r_mout9 <= w_mout9;
        r_mout10 <= w_mout10;
        r_mout11 <= w_mout11;
        r_mout12 <= w_mout12;
```

```
                r_mout13 <= w_mout13;
                r_mout14 <= w_mout14;
                r_mout15 <= w_mout15;
        end

        always @(*)
        begin
                w_selmut0 <= (0==w_mutout[3:0])?1:0;
                w_selmut1 <= (1==w_mutout[3:0])?1:0;
                w_selmut2 <= (2==w_mutout[3:0])?1:0;
                w_selmut3 <= (3==w_mutout[3:0])?1:0;
                w_selmut4 <= (4==w_mutout[3:0])?1:0;
                w_selmut5 <= (5==w_mutout[3:0])?1:0;
                w_selmut6 <= (6==w_mutout[3:0])?1:0;
                w_selmut7 <= (7==w_mutout[3:0])?1:0;
                w_selmut8 <= (8==w_mutout[3:0])?1:0;
                w_selmut9 <= (9==w_mutout[3:0])?1:0;
                w_selmut10 <= (10==w_mutout[3:0])?1:0;
                w_selmut11 <= (11==w_mutout[3:0])?1:0;
                w_selmut12 <= (12==w_mutout[3:0])?1:0;
                w_selmut13 <= (13==w_mutout[3:0])?1:0;
                w_selmut14 <= (14==w_mutout[3:0])?1:0;
                w_selmut15 <= (15==w_mutout[3:0])?1:0;
                w_muted0 = w_out0 ^ w_mutout[17:4];
                w_muted1 = w_out1 ^ w_mutout[17:4];
                w_muted2 = w_out2 ^ w_mutout[17:4];
                w_muted3 = w_out3 ^ w_mutout[17:4];
                w_muted4 = w_out4 ^ w_mutout[17:4];
                w_muted5 = w_out5 ^ w_mutout[17:4];
                w_muted6 = w_out6 ^ w_mutout[17:4];
                w_muted7 = w_out7 ^ w_mutout[17:4];
                w_muted8 = w_out8 ^ w_mutout[17:4];
                w_muted9 = w_out9 ^ w_mutout[17:4];
                w_muted10 = w_out10 ^ w_mutout[17:4];
                w_muted11 = w_out11 ^ w_mutout[17:4];
                w_muted12 = w_out12 ^ w_mutout[17:4];
                w_muted13 = w_out13 ^ w_mutout[17:4];
                w_muted14 = w_out14 ^ w_mutout[17:4];
                w_muted15 = w_out15 ^ w_mutout[17:4];
        end
endmodule
```

```
module swapperN(a, b, lt, aprime, bprime);
	parameter n=16;
	input [n-1:0] a, b;
	input lt;
	output [n-1:0] aprime, bprime;
	reg [n-1:0] aprime, bprime;
	always @(a, b, lt)
	begin
		if (lt)
		begin
			aprime <=b;
			bprime <=a;
		end
		else
		begin
			aprime <=a;
			bprime <=b;
		end
	end
endmodule


module dff_NB14(d, CLK, q);
	parameter n=14;
	input [n-1:0] d;
	input CLK;
	output [n-1:0] q;
	reg [n-1:0] q;
	always @(posedge CLK)
		q <= d;
endmodule

module dff_NB8(d, CLK, q);
	parameter n=8;
	input [n-1:0] d;
	input CLK;
	output [n-1:0] q;
	reg [n-1:0] q;
	always @(posedge CLK)
		q <= d;
endmodule
```

```
module dff_N(d, CLK, q);
	parameter n=16;
	input [n-1:0] d;
	input CLK;
	output [n-1:0] q;
	reg [n-1:0] q;

	always @(posedge CLK)
		q <= d;
endmodule

module compare_lt(a, b, lt);
	parameter n=8;
	input [n-1:0] a, b;
	output lt;
	reg lt;

	always @(a,b)
		if (a<b)
			lt<=1'b1;
		else
			lt<=1'b0;
endmodule

module sort2(a, b, aprime, bprime);
	//Configuration of inputs {8bit index, 8bit fitness value}
	//parameter n=16;
	//parameter k=8;
	parameter wid_TT=14;
	parameter wid_fit=8;
	input [wid_TT+wid_fit-1:0] a, b;
	output [wid_TT+wid_fit-1:0] aprime, bprime;
	wire [wid_TT+wid_fit-1:0] aprime, bprime;
	wire lt;

	compare_lt comp(a[wid_fit-1:0], b[wid_fit-1:0], lt);

	defparam s.n=wid_TT+wid_fit;
	swapperN s(a, b, lt, aprime, bprime);
endmodule

module sort4(a,
```

```
          b,
          c,
          CLK,
          d,
          aprime,
          bprime,
          cprime,
          dprime);

    parameter wid_TT=14;
    parameter wid_fit=8;

 input [wid_TT+wid_fit-1:0] a;
 input [wid_TT+wid_fit-1:0] b;
 input [wid_TT+wid_fit-1:0] c;
 input CLK;
 input [wid_TT+wid_fit-1:0] d;
output [wid_TT+wid_fit-1:0] aprime;
output [wid_TT+wid_fit-1:0] bprime;
output [wid_TT+wid_fit-1:0] cprime;
output [wid_TT+wid_fit-1:0] dprime;

wire [wid_TT+wid_fit-1:0] XLXN_1;
wire [wid_TT+wid_fit-1:0] XLXN_2;
wire [wid_TT+wid_fit-1:0] XLXN_3;
wire [wid_TT+wid_fit-1:0] XLXN_4;
wire [wid_TT+wid_fit-1:0] XLXN_9;
wire [wid_TT+wid_fit-1:0] XLXN_10;
wire [wid_TT+wid_fit-1:0] XLXN_11;
wire [wid_TT+wid_fit-1:0] XLXN_12;
wire [wid_TT+wid_fit-1:0] XLXN_13;
wire [wid_TT+wid_fit-1:0] XLXN_14;
wire [wid_TT+wid_fit-1:0] XLXN_15;
wire [wid_TT+wid_fit-1:0] XLXN_16;
wire [wid_TT+wid_fit-1:0] XLXN_36;
wire [wid_TT+wid_fit-1:0] XLXN_37;
wire [wid_TT+wid_fit-1:0] XLXN_40;
wire [wid_TT+wid_fit-1:0] XLXN_41;
wire [wid_TT+wid_fit-1:0] XLXN_42;
wire [wid_TT+wid_fit-1:0] XLXN_43;

    defparam XLXI_1.n=wid_TT+wid_fit;
```

```
dff_N XLXI_1 (.CLK(CLK),
             .d(XLXN_1[wid_TT+wid_fit-1:0]),
             .q(XLXN_9[wid_TT+wid_fit-1:0]));

    defparam XLXI_2.wid_TT=wid_TT;
    defparam XLXI_2.wid_fit=wid_fit;
sort2 XLXI_2 (.a(a[wid_TT+wid_fit-1:0]),
              .b(b[wid_TT+wid_fit-1:0]),
              .aprime(XLXN_1[wid_TT+wid_fit-1:0]),
              .bprime(XLXN_2[wid_TT+wid_fit-1:0]));

    defparam XLXI_3.wid_TT=wid_TT;
    defparam XLXI_3.wid_fit=wid_fit;
sort2 XLXI_3 (.a(c[wid_TT+wid_fit-1:0]),
              .b(d[wid_TT+wid_fit-1:0]),
              .aprime(XLXN_3[wid_TT+wid_fit-1:0]),
              .bprime(XLXN_4[wid_TT+wid_fit-1:0]));

defparam XLXI_5.n=wid_TT+wid_fit;
dff_N XLXI_5 (.CLK(CLK),
             .d(XLXN_2[wid_TT+wid_fit-1:0]),
             .q(XLXN_11[wid_TT+wid_fit-1:0]));

defparam XLXI_6.n=wid_TT+wid_fit;
dff_N XLXI_6 (.CLK(CLK),
             .d(XLXN_3[wid_TT+wid_fit-1:0]),
             .q(XLXN_10[wid_TT+wid_fit-1:0]));

defparam XLXI_7.n=wid_TT+wid_fit;
dff_N XLXI_7 (.CLK(CLK),
             .d(XLXN_4[wid_TT+wid_fit-1:0]),
             .q(XLXN_12[wid_TT+wid_fit-1:0]));

    defparam XLXI_8.wid_TT=wid_TT;
    defparam XLXI_8.wid_fit=wid_fit;
sort2 XLXI_8 (.a(XLXN_9[wid_TT+wid_fit-1:0]),
              .b(XLXN_10[wid_TT+wid_fit-1:0]),
              .aprime(XLXN_13[wid_TT+wid_fit-1:0]),
              .bprime(XLXN_14[wid_TT+wid_fit-1:0]));

    defparam XLXI_9.wid_TT=wid_TT;
    defparam XLXI_9.wid_fit=wid_fit;
```

```
 sort2 XLXI_9 (.a(XLXN_11[wid_TT+wid_fit-1:0]),
               .b(XLXN_12[wid_TT+wid_fit-1:0]),
               .aprime(XLXN_15[wid_TT+wid_fit-1:0]),
               .bprime(XLXN_16[wid_TT+wid_fit-1:0]));

 defparam XLXI_10.n=wid_TT+wid_fit;
 dff_N XLXI_10 (.CLK(CLK),
               .d(XLXN_13[wid_TT+wid_fit-1:0]),
               .q(XLXN_40[wid_TT+wid_fit-1:0]));

 defparam XLXI_11.n=wid_TT+wid_fit;
 dff_N XLXI_11 (.CLK(CLK),
               .d(XLXN_14[wid_TT+wid_fit-1:0]),
               .q(XLXN_36[wid_TT+wid_fit-1:0]));

 defparam XLXI_12.n=wid_TT+wid_fit;
 dff_N XLXI_12 (.CLK(CLK),
               .d(XLXN_15[wid_TT+wid_fit-1:0]),
               .q(XLXN_37[wid_TT+wid_fit-1:0]));

  defparam XLXI_13.n=wid_TT+wid_fit;
dff_N XLXI_13 (.CLK(CLK),
               .d(XLXN_16[wid_TT+wid_fit-1:0]),
               .q(XLXN_43[wid_TT+wid_fit-1:0]));

     defparam XLXI_14.wid_TT=wid_TT;
     defparam XLXI_14.wid_fit=wid_fit;
 sort2 XLXI_14 (.a(XLXN_36[wid_TT+wid_fit-1:0]),
                .b(XLXN_37[wid_TT+wid_fit-1:0]),
                .aprime(XLXN_41[wid_TT+wid_fit-1:0]),
                .bprime(XLXN_42[wid_TT+wid_fit-1:0]));

 defparam XLXI_31.n=wid_TT+wid_fit;
 dff_N XLXI_31 (.CLK(CLK),
               .d(XLXN_40[wid_TT+wid_fit-1:0]),
               .q(aprime[wid_TT+wid_fit-1:0]));

 defparam XLXI_32.n=wid_TT+wid_fit;
 dff_N XLXI_32 (.CLK(CLK),
               .d(XLXN_41[wid_TT+wid_fit-1:0]),
               .q(bprime[wid_TT+wid_fit-1:0]));
```

```
   defparam XLXI_33.n=wid_TT+wid_fit;
   dff_N XLXI_33 (.CLK(CLK),
                  .d(XLXN_42[wid_TT+wid_fit-1:0]),
                  .q(cprime[wid_TT+wid_fit-1:0]));

   defparam XLXI_34.n=wid_TT+wid_fit;
   dff_N XLXI_34 (.CLK(CLK),
                  .d(XLXN_43[wid_TT+wid_fit-1:0]),
                  .q(dprime[wid_TT+wid_fit-1:0]));
endmodule

module sort8(a,
             b,
             c,
             CLK,
             d,
             e,
             f,
             g,
             h,
             aprime,
             bprime,
             cprime,
             dprime,
             eprime,
             fprime,
             gprime,
             hprime);

        parameter wid_TT=14;
        parameter wid_fit=8;

    input [wid_TT+wid_fit-1:0] a;
    input [wid_TT+wid_fit-1:0] b;
    input [wid_TT+wid_fit-1:0] c;
    input CLK;
    input [wid_TT+wid_fit-1:0] d;
    input [wid_TT+wid_fit-1:0] e;
    input [wid_TT+wid_fit-1:0] f;
    input [wid_TT+wid_fit-1:0] g;
    input [wid_TT+wid_fit-1:0] h;
   output [wid_TT+wid_fit-1:0] aprime;
```

```
output [wid_TT+wid_fit-1:0] bprime;
output [wid_TT+wid_fit-1:0] cprime;
output [wid_TT+wid_fit-1:0] dprime;
output [wid_TT+wid_fit-1:0] eprime;
output [wid_TT+wid_fit-1:0] fprime;
output [wid_TT+wid_fit-1:0] gprime;
output [wid_TT+wid_fit-1:0] hprime;

wire [wid_TT+wid_fit-1:0] XLXN_11;
wire [wid_TT+wid_fit-1:0] XLXN_12;
wire [wid_TT+wid_fit-1:0] XLXN_13;
wire [wid_TT+wid_fit-1:0] XLXN_14;
wire [wid_TT+wid_fit-1:0] XLXN_15;
wire [wid_TT+wid_fit-1:0] XLXN_16;
wire [wid_TT+wid_fit-1:0] XLXN_17;
wire [wid_TT+wid_fit-1:0] XLXN_18;
wire [wid_TT+wid_fit-1:0] XLXN_19;
wire [wid_TT+wid_fit-1:0] XLXN_20;
wire [wid_TT+wid_fit-1:0] XLXN_21;
wire [wid_TT+wid_fit-1:0] XLXN_22;
wire [wid_TT+wid_fit-1:0] XLXN_23;
wire [wid_TT+wid_fit-1:0] XLXN_24;
wire [wid_TT+wid_fit-1:0] XLXN_25;
wire [wid_TT+wid_fit-1:0] XLXN_26;
wire [wid_TT+wid_fit-1:0] XLXN_28;
wire [wid_TT+wid_fit-1:0] XLXN_29;
wire [wid_TT+wid_fit-1:0] XLXN_30;
wire [wid_TT+wid_fit-1:0] XLXN_31;
wire [wid_TT+wid_fit-1:0] XLXN_32;
wire [wid_TT+wid_fit-1:0] XLXN_33;
wire [wid_TT+wid_fit-1:0] XLXN_34;
wire [wid_TT+wid_fit-1:0] XLXN_35;
wire [wid_TT+wid_fit-1:0] XLXN_36;
wire [wid_TT+wid_fit-1:0] XLXN_37;
wire [wid_TT+wid_fit-1:0] XLXN_38;
wire [wid_TT+wid_fit-1:0] XLXN_39;
wire [wid_TT+wid_fit-1:0] XLXN_46;
wire [wid_TT+wid_fit-1:0] XLXN_47;
wire [wid_TT+wid_fit-1:0] XLXN_48;
wire [wid_TT+wid_fit-1:0] XLXN_49;
wire [wid_TT+wid_fit-1:0] XLXN_50;
wire [wid_TT+wid_fit-1:0] XLXN_51;
```

```
wire [wid_TT+wid_fit-1:0] XLXN_60;
wire [wid_TT+wid_fit-1:0] XLXN_61;
wire [wid_TT+wid_fit-1:0] XLXN_62;
wire [wid_TT+wid_fit-1:0] XLXN_63;
wire [wid_TT+wid_fit-1:0] XLXN_64;
wire [wid_TT+wid_fit-1:0] XLXN_65;
wire [wid_TT+wid_fit-1:0] XLXN_66;
wire [wid_TT+wid_fit-1:0] XLXN_67;

    defparam XLXI_1.wid_TT=wid_TT;
    defparam XLXI_1.wid_fit=wid_fit;
sort4 XLXI_1 (.a(a[wid_TT+wid_fit-1:0]),
              .b(b[wid_TT+wid_fit-1:0]),
              .c(c[wid_TT+wid_fit-1:0]),
              .CLK(CLK),
              .d(d[wid_TT+wid_fit-1:0]),
              .aprime(XLXN_11[wid_TT+wid_fit-1:0]),
              .bprime(XLXN_15[wid_TT+wid_fit-1:0]),
              .cprime(XLXN_13[wid_TT+wid_fit-1:0]),
              .dprime(XLXN_17[wid_TT+wid_fit-1:0]));

    defparam XLXI_2.wid_TT=wid_TT;
    defparam XLXI_2.wid_fit=wid_fit;
sort4 XLXI_2 (.a(e[wid_TT+wid_fit-1:0]),
              .b(f[wid_TT+wid_fit-1:0]),
              .c(g[wid_TT+wid_fit-1:0]),
              .CLK(CLK),
              .d(h[wid_TT+wid_fit-1:0]),
              .aprime(XLXN_12[wid_TT+wid_fit-1:0]),
              .bprime(XLXN_16[wid_TT+wid_fit-1:0]),
              .cprime(XLXN_14[wid_TT+wid_fit-1:0]),
              .dprime(XLXN_18[wid_TT+wid_fit-1:0]));

    defparam XLXI_3.wid_TT=wid_TT;
    defparam XLXI_3.wid_fit=wid_fit;
sort2 XLXI_3 (.a(XLXN_11[wid_TT+wid_fit-1:0]),
              .b(XLXN_12[wid_TT+wid_fit-1:0]),
              .aprime(XLXN_19[wid_TT+wid_fit-1:0]),
              .bprime(XLXN_20[wid_TT+wid_fit-1:0]));

    defparam XLXI_4.wid_TT=wid_TT;
    defparam XLXI_4.wid_fit=wid_fit;
```

```
sort2 XLXI_4 (.a(XLXN_13[wid_TT+wid_fit-1:0]),
              .b(XLXN_14[wid_TT+wid_fit-1:0]),
              .aprime(XLXN_21[wid_TT+wid_fit-1:0]),
              .bprime(XLXN_22[wid_TT+wid_fit-1:0]));

    defparam XLXI_5.wid_TT=wid_TT;
    defparam XLXI_5.wid_fit=wid_fit;
sort2 XLXI_5 (.a(XLXN_15[wid_TT+wid_fit-1:0]),
              .b(XLXN_16[wid_TT+wid_fit-1:0]),
              .aprime(XLXN_23[wid_TT+wid_fit-1:0]),
              .bprime(XLXN_24[wid_TT+wid_fit-1:0]));

    defparam XLXI_6.wid_TT=wid_TT;
    defparam XLXI_6.wid_fit=wid_fit;
sort2 XLXI_6 (.a(XLXN_17[wid_TT+wid_fit-1:0]),
              .b(XLXN_18[wid_TT+wid_fit-1:0]),
              .aprime(XLXN_25[wid_TT+wid_fit-1:0]),
              .bprime(XLXN_26[wid_TT+wid_fit-1:0]));

    defparam XLXI_7.n=wid_TT+wid_fit;
dff_N XLXI_7 (.CLK(CLK),
             .d(XLXN_19[wid_TT+wid_fit-1:0]),
             .q(XLXN_32[wid_TT+wid_fit-1:0]));

    defparam XLXI_8.n=wid_TT+wid_fit;
dff_N XLXI_8 (.CLK(CLK),
             .d(XLXN_23[wid_TT+wid_fit-1:0]),
             .q(XLXN_33[wid_TT+wid_fit-1:0]));

    defparam XLXI_9.n=wid_TT+wid_fit;
dff_N XLXI_9 (.CLK(CLK),
             .d(XLXN_21[wid_TT+wid_fit-1:0]),
             .q(XLXN_28[wid_TT+wid_fit-1:0]));

    defparam XLXI_10.n=wid_TT+wid_fit;
dff_N XLXI_10 (.CLK(CLK),
              .d(XLXN_25[wid_TT+wid_fit-1:0]),
              .q(XLXN_30[wid_TT+wid_fit-1:0]));

    defparam XLXI_11.n=wid_TT+wid_fit;
dff_N XLXI_11 (.CLK(CLK),
              .d(XLXN_20[wid_TT+wid_fit-1:0]),
```

```
              .q(XLXN_29[wid_TT+wid_fit-1:0]));

    defparam XLXI_12.n=wid_TT+wid_fit;
dff_N XLXI_12 (.CLK(CLK),
              .d(XLXN_24[wid_TT+wid_fit-1:0]),
              .q(XLXN_31[wid_TT+wid_fit-1:0]));

    defparam XLXI_13.n=wid_TT+wid_fit;
dff_N XLXI_13 (.CLK(CLK),
              .d(XLXN_22[wid_TT+wid_fit-1:0]),
              .q(XLXN_38[wid_TT+wid_fit-1:0]));

    defparam XLXI_14.n=wid_TT+wid_fit;
dff_N XLXI_14 (.CLK(CLK),
              .d(XLXN_26[wid_TT+wid_fit-1:0]),
              .q(XLXN_39[wid_TT+wid_fit-1:0]));

    defparam XLXI_23.wid_TT=wid_TT;
    defparam XLXI_23.wid_fit=wid_fit;
sort2 XLXI_23 (.a(XLXN_28[wid_TT+wid_fit-1:0]),
               .b(XLXN_29[wid_TT+wid_fit-1:0]),
               .aprime(XLXN_34[wid_TT+wid_fit-1:0]),
               .bprime(XLXN_35[wid_TT+wid_fit-1:0]));

    defparam XLXI_24.wid_TT=wid_TT;
    defparam XLXI_24.wid_fit=wid_fit;
sort2 XLXI_24 (.a(XLXN_30[wid_TT+wid_fit-1:0]),
               .b(XLXN_31[wid_TT+wid_fit-1:0]),
               .aprime(XLXN_36[wid_TT+wid_fit-1:0]),
               .bprime(XLXN_37[wid_TT+wid_fit-1:0]));

    defparam XLXI_25.n=wid_TT+wid_fit;
dff_N XLXI_25 (.CLK(CLK),
              .d(XLXN_32[wid_TT+wid_fit-1:0]),
              .q(XLXN_60[wid_TT+wid_fit-1:0]));

    defparam XLXI_26.n=wid_TT+wid_fit;
dff_N XLXI_26 (.CLK(CLK),
              .d(XLXN_33[wid_TT+wid_fit-1:0]),
              .q(XLXN_46[wid_TT+wid_fit-1:0]));

    defparam XLXI_27.n=wid_TT+wid_fit;
```

```
dff_N XLXI_27 (.CLK(CLK),
               .d(XLXN_34[wid_TT+wid_fit-1:0]),
               .q(XLXN_47[wid_TT+wid_fit-1:0]));

    defparam XLXI_29.n=wid_TT+wid_fit;
dff_N XLXI_29 (.CLK(CLK),
               .d(XLXN_36[wid_TT+wid_fit-1:0]),
               .q(XLXN_48[wid_TT+wid_fit-1:0]));

    defparam XLXI_30.n=wid_TT+wid_fit;
dff_N XLXI_30 (.CLK(CLK),
               .d(XLXN_35[wid_TT+wid_fit-1:0]),
               .q(XLXN_49[wid_TT+wid_fit-1:0]));

    defparam XLXI_31.n=wid_TT+wid_fit;
dff_N XLXI_31 (.CLK(CLK),
               .d(XLXN_37[wid_TT+wid_fit-1:0]),
               .q(XLXN_50[wid_TT+wid_fit-1:0]));

    defparam XLXI_32.n=wid_TT+wid_fit;
dff_N XLXI_32 (.CLK(CLK),
               .d(XLXN_38[wid_TT+wid_fit-1:0]),
               .q(XLXN_51[wid_TT+wid_fit-1:0]));

    defparam XLXI_33.n=wid_TT+wid_fit;
dff_N XLXI_33 (.CLK(CLK),
               .d(XLXN_39[wid_TT+wid_fit-1:0]),
               .q(XLXN_67[wid_TT+wid_fit-1:0]));

    defparam XLXI_34.wid_TT=wid_TT;
    defparam XLXI_34.wid_fit=wid_fit;
sort2 XLXI_34 (.a(XLXN_46[wid_TT+wid_fit-1:0]),
               .b(XLXN_47[wid_TT+wid_fit-1:0]),
               .aprime(XLXN_61[wid_TT+wid_fit-1:0]),
               .bprime(XLXN_62[wid_TT+wid_fit-1:0]));

    defparam XLXI_35.wid_TT=wid_TT;
    defparam XLXI_35.wid_fit=wid_fit;
sort2 XLXI_35 (.a(XLXN_48[wid_TT+wid_fit-1:0]),
               .b(XLXN_49[wid_TT+wid_fit-1:0]),
               .aprime(XLXN_63[wid_TT+wid_fit-1:0]),
               .bprime(XLXN_64[wid_TT+wid_fit-1:0]));
```

```
    defparam XLXI_36.wid_TT=wid_TT;
    defparam XLXI_36.wid_fit=wid_fit;
sort2 XLXI_36 (.a(XLXN_50[wid_TT+wid_fit-1:0]),
               .b(XLXN_51[wid_TT+wid_fit-1:0]),
               .aprime(XLXN_65[wid_TT+wid_fit-1:0]),
               .bprime(XLXN_66[wid_TT+wid_fit-1:0]));

    defparam XLXI_45.n=wid_TT+wid_fit;
dff_N XLXI_45 (.CLK(CLK),
              .d(XLXN_60[wid_TT+wid_fit-1:0]),
              .q(aprime[wid_TT+wid_fit-1:0]));

    defparam XLXI_46.n=wid_TT+wid_fit;
dff_N XLXI_46 (.CLK(CLK),
              .d(XLXN_61[wid_TT+wid_fit-1:0]),
              .q(bprime[wid_TT+wid_fit-1:0]));

    defparam XLXI_47.n=wid_TT+wid_fit;
dff_N XLXI_47 (.CLK(CLK),
              .d(XLXN_62[wid_TT+wid_fit-1:0]),
              .q(cprime[wid_TT+wid_fit-1:0]));

    defparam XLXI_48.n=wid_TT+wid_fit;
dff_N XLXI_48 (.CLK(CLK),
              .d(XLXN_63[wid_TT+wid_fit-1:0]),
              .q(dprime[wid_TT+wid_fit-1:0]));

    defparam XLXI_49.n=wid_TT+wid_fit;
dff_N XLXI_49 (.CLK(CLK),
              .d(XLXN_64[wid_TT+wid_fit-1:0]),
              .q(eprime[wid_TT+wid_fit-1:0]));

    defparam XLXI_50.n=wid_TT+wid_fit;
dff_N XLXI_50 (.CLK(CLK),
              .d(XLXN_65[wid_TT+wid_fit-1:0]),
              .q(fprime[wid_TT+wid_fit-1:0]));

    defparam XLXI_51.n=wid_TT+wid_fit;
dff_N XLXI_51 (.CLK(CLK),
              .d(XLXN_66[wid_TT+wid_fit-1:0]),
              .q(gprime[wid_TT+wid_fit-1:0]));
```

```
            defparam XLXI_52.n=wid_TT+wid_fit;
          dff_N XLXI_52 (.CLK(CLK),
                        .d(XLXN_67[wid_TT+wid_fit-1:0]),
                        .q(hprime[wid_TT+wid_fit-1:0]));
       endmodule


       module   sort(a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p,CLK, aprime, bprime,
cprime, dprime, eprime, fprime, gprime, hprime, iprime, jprime, kprime, lprime, mprime,
nprime, oprime, pprime);


            parameter wid_TT=14;
            parameter wid_fit=8;


           input [wid_TT+wid_fit-1:0] a;
           input [wid_TT+wid_fit-1:0] b;
           input [wid_TT+wid_fit-1:0] c;
           //input CLK;
           input [wid_TT+wid_fit-1:0] d;
           input [wid_TT+wid_fit-1:0] e;
           input [wid_TT+wid_fit-1:0] f;
           input [wid_TT+wid_fit-1:0] g;
           input [wid_TT+wid_fit-1:0] h;
           input [wid_TT+wid_fit-1:0] i;
           input [wid_TT+wid_fit-1:0] j;
           input [wid_TT+wid_fit-1:0] k;
           input [wid_TT+wid_fit-1:0] l;
           input [wid_TT+wid_fit-1:0] m;
           input [wid_TT+wid_fit-1:0] n;
           input [wid_TT+wid_fit-1:0] o;
           input [wid_TT+wid_fit-1:0] p;
           input CLK /* synthesis syn_noclockbuf=1 syn_maxfan=100000 */;
          output [wid_TT+wid_fit-1:0] aprime;
          output [wid_TT+wid_fit-1:0] bprime;
          output [wid_TT+wid_fit-1:0] cprime;
          output [wid_TT+wid_fit-1:0] dprime;
          output [wid_TT+wid_fit-1:0] eprime;
          output [wid_TT+wid_fit-1:0] fprime;
          output [wid_TT+wid_fit-1:0] gprime;
          output [wid_TT+wid_fit-1:0] hprime;
          output [wid_TT+wid_fit-1:0] iprime;
          output [wid_TT+wid_fit-1:0] jprime;
```

```
output [wid_TT+wid_fit-1:0] kprime;
output [wid_TT+wid_fit-1:0] lprime;
output [wid_TT+wid_fit-1:0] mprime;
output [wid_TT+wid_fit-1:0] nprime;
output [wid_TT+wid_fit-1:0] oprime;
output [wid_TT+wid_fit-1:0] pprime;

wire [wid_TT+wid_fit-1:0] XLXN_17;
wire [wid_TT+wid_fit-1:0] XLXN_31;
wire [wid_TT+wid_fit-1:0] XLXN_49;
wire [wid_TT+wid_fit-1:0] XLXN_50;
wire [wid_TT+wid_fit-1:0] XLXN_51;
wire [wid_TT+wid_fit-1:0] XLXN_52;
wire [wid_TT+wid_fit-1:0] XLXN_53;
wire [wid_TT+wid_fit-1:0] XLXN_54;
wire [wid_TT+wid_fit-1:0] XLXN_55;
wire [wid_TT+wid_fit-1:0] XLXN_56;
wire [wid_TT+wid_fit-1:0] XLXN_57;
wire [wid_TT+wid_fit-1:0] XLXN_58;
wire [wid_TT+wid_fit-1:0] XLXN_59;
wire [wid_TT+wid_fit-1:0] XLXN_60;
wire [wid_TT+wid_fit-1:0] XLXN_61;
wire [wid_TT+wid_fit-1:0] XLXN_62;
wire [wid_TT+wid_fit-1:0] XLXN_63;
wire [wid_TT+wid_fit-1:0] XLXN_64;
wire [wid_TT+wid_fit-1:0] XLXN_65;
wire [wid_TT+wid_fit-1:0] XLXN_66;
wire [wid_TT+wid_fit-1:0] XLXN_67;
wire [wid_TT+wid_fit-1:0] XLXN_68;
wire [wid_TT+wid_fit-1:0] XLXN_69;
wire [wid_TT+wid_fit-1:0] XLXN_70;
wire [wid_TT+wid_fit-1:0] XLXN_71;
wire [wid_TT+wid_fit-1:0] XLXN_72;
wire [wid_TT+wid_fit-1:0] XLXN_73;
wire [wid_TT+wid_fit-1:0] XLXN_74;
wire [wid_TT+wid_fit-1:0] XLXN_75;
wire [wid_TT+wid_fit-1:0] XLXN_76;
wire [wid_TT+wid_fit-1:0] XLXN_77;
wire [wid_TT+wid_fit-1:0] XLXN_78;
wire [wid_TT+wid_fit-1:0] XLXN_278;
wire [wid_TT+wid_fit-1:0] XLXN_424;
wire [wid_TT+wid_fit-1:0] XLXN_452;
```

```
wire [wid_TT+wid_fit-1:0] XLXN_453;
wire [wid_TT+wid_fit-1:0] XLXN_454;
wire [wid_TT+wid_fit-1:0] XLXN_455;
wire [wid_TT+wid_fit-1:0] XLXN_456;
wire [wid_TT+wid_fit-1:0] XLXN_457;
wire [wid_TT+wid_fit-1:0] XLXN_458;
wire [wid_TT+wid_fit-1:0] XLXN_459;
wire [wid_TT+wid_fit-1:0] XLXN_460;
wire [wid_TT+wid_fit-1:0] XLXN_461;
wire [wid_TT+wid_fit-1:0] XLXN_462;
wire [wid_TT+wid_fit-1:0] XLXN_463;
wire [wid_TT+wid_fit-1:0] XLXN_464;
wire [wid_TT+wid_fit-1:0] XLXN_465;
wire [wid_TT+wid_fit-1:0] XLXN_466;
wire [wid_TT+wid_fit-1:0] XLXN_467;
wire [wid_TT+wid_fit-1:0] XLXN_468;
wire [wid_TT+wid_fit-1:0] XLXN_469;
wire [wid_TT+wid_fit-1:0] XLXN_470;
wire [wid_TT+wid_fit-1:0] XLXN_471;
wire [wid_TT+wid_fit-1:0] XLXN_472;
wire [wid_TT+wid_fit-1:0] XLXN_473;
wire [wid_TT+wid_fit-1:0] XLXN_474;
wire [wid_TT+wid_fit-1:0] XLXN_475;
wire [wid_TT+wid_fit-1:0] XLXN_476;
wire [wid_TT+wid_fit-1:0] XLXN_477;
wire [wid_TT+wid_fit-1:0] XLXN_478;
wire [wid_TT+wid_fit-1:0] XLXN_479;
wire [wid_TT+wid_fit-1:0] XLXN_480;
wire [wid_TT+wid_fit-1:0] XLXN_481;
wire [wid_TT+wid_fit-1:0] XLXN_482;
wire [wid_TT+wid_fit-1:0] XLXN_483;
wire [wid_TT+wid_fit-1:0] XLXN_484;
wire [wid_TT+wid_fit-1:0] XLXN_485;
wire [wid_TT+wid_fit-1:0] XLXN_486;
wire [wid_TT+wid_fit-1:0] XLXN_487;
wire [wid_TT+wid_fit-1:0] XLXN_488;
wire [wid_TT+wid_fit-1:0] XLXN_489;
wire [wid_TT+wid_fit-1:0] XLXN_490;
wire [wid_TT+wid_fit-1:0] XLXN_491;
wire [wid_TT+wid_fit-1:0] XLXN_492;
wire [wid_TT+wid_fit-1:0] XLXN_493;
wire [wid_TT+wid_fit-1:0] XLXN_494;
```

```
wire [wid_TT+wid_fit-1:0] XLXN_495;
wire [wid_TT+wid_fit-1:0] XLXN_496;
wire [wid_TT+wid_fit-1:0] XLXN_497;
wire [wid_TT+wid_fit-1:0] XLXN_498;
wire [wid_TT+wid_fit-1:0] XLXN_499;
wire [wid_TT+wid_fit-1:0] XLXN_500;
wire [wid_TT+wid_fit-1:0] XLXN_501;
wire [wid_TT+wid_fit-1:0] XLXN_502;
wire [wid_TT+wid_fit-1:0] XLXN_503;
wire [wid_TT+wid_fit-1:0] XLXN_515;
wire [wid_TT+wid_fit-1:0] XLXN_517;
wire [wid_TT+wid_fit-1:0] XLXN_518;
wire [wid_TT+wid_fit-1:0] XLXN_519;
wire [wid_TT+wid_fit-1:0] XLXN_520;
wire [wid_TT+wid_fit-1:0] XLXN_521;
wire [wid_TT+wid_fit-1:0] XLXN_522;
wire [wid_TT+wid_fit-1:0] XLXN_524;
wire [wid_TT+wid_fit-1:0] XLXN_525;
wire [wid_TT+wid_fit-1:0] XLXN_526;
wire [wid_TT+wid_fit-1:0] XLXN_527;
wire [wid_TT+wid_fit-1:0] XLXN_528;
wire [wid_TT+wid_fit-1:0] XLXN_529;
wire [wid_TT+wid_fit-1:0] XLXN_530;
wire [wid_TT+wid_fit-1:0] XLXN_531;
wire [wid_TT+wid_fit-1:0] XLXN_532;
wire [wid_TT+wid_fit-1:0] XLXN_533;
wire [wid_TT+wid_fit-1:0] XLXN_534;
wire [wid_TT+wid_fit-1:0] XLXN_535;
wire [wid_TT+wid_fit-1:0] XLXN_536;
wire [wid_TT+wid_fit-1:0] XLXN_537;
wire [wid_TT+wid_fit-1:0] XLXN_538;
wire [wid_TT+wid_fit-1:0] XLXN_539;
wire [wid_TT+wid_fit-1:0] XLXN_540;
wire [wid_TT+wid_fit-1:0] XLXN_541;
wire [wid_TT+wid_fit-1:0] XLXN_542;
wire [wid_TT+wid_fit-1:0] XLXN_543;
wire [wid_TT+wid_fit-1:0] XLXN_544;

    defparam XLXI_1.wid_TT=wid_TT;
    defparam XLXI_1.wid_fit=wid_fit;
sort8 XLXI_1 (.a(a[wid_TT+wid_fit-1:0]),
              .b(b[wid_TT+wid_fit-1:0]),
```

```
                .c(c[wid_TT+wid_fit-1:0]),
                .CLK(CLK),
                .d(d[wid_TT+wid_fit-1:0]),
                .e(e[wid_TT+wid_fit-1:0]),
                .f(f[wid_TT+wid_fit-1:0]),
                .g(g[wid_TT+wid_fit-1:0]),
                .h(h[wid_TT+wid_fit-1:0]),
                .aprime(XLXN_17[wid_TT+wid_fit-1:0]),
                .bprime(XLXN_51[wid_TT+wid_fit-1:0]),
                .cprime(XLXN_55[wid_TT+wid_fit-1:0]),
                .dprime(XLXN_59[wid_TT+wid_fit-1:0]),
                .eprime(XLXN_63[wid_TT+wid_fit-1:0]),
                .fprime(XLXN_67[wid_TT+wid_fit-1:0]),
                .gprime(XLXN_71[wid_TT+wid_fit-1:0]),
                .hprime(XLXN_75[wid_TT+wid_fit-1:0]));

    defparam XLXI_2.wid_TT=wid_TT;
    defparam XLXI_2.wid_fit=wid_fit;
sort8 XLXI_2 (.a(i[wid_TT+wid_fit-1:0]),
                .b(j[wid_TT+wid_fit-1:0]),
                .c(k[wid_TT+wid_fit-1:0]),
                .CLK(CLK),
                .d(l[wid_TT+wid_fit-1:0]),
                .e(m[wid_TT+wid_fit-1:0]),
                .f(n[wid_TT+wid_fit-1:0]),
                .g(o[wid_TT+wid_fit-1:0]),
                .h(p[wid_TT+wid_fit-1:0]),
                .aprime(XLXN_31[wid_TT+wid_fit-1:0]),
                .bprime(XLXN_52[wid_TT+wid_fit-1:0]),
                .cprime(XLXN_56[wid_TT+wid_fit-1:0]),
                .dprime(XLXN_60[wid_TT+wid_fit-1:0]),
                .eprime(XLXN_64[wid_TT+wid_fit-1:0]),
                .fprime(XLXN_68[wid_TT+wid_fit-1:0]),
                .gprime(XLXN_72[wid_TT+wid_fit-1:0]),
                .hprime(XLXN_76[wid_TT+wid_fit-1:0]));

    defparam XLXI_3.wid_TT=wid_TT;
    defparam XLXI_3.wid_fit=wid_fit;
sort2 XLXI_3 (.a(XLXN_17[wid_TT+wid_fit-1:0]),
                .b(XLXN_31[wid_TT+wid_fit-1:0]),
                .aprime(XLXN_49[wid_TT+wid_fit-1:0]),
                .bprime(XLXN_50[wid_TT+wid_fit-1:0]));
```

```
    defparam XLXI_4.wid_TT=wid_TT;
    defparam XLXI_4.wid_fit=wid_fit;
sort2 XLXI_4 (.a(XLXN_51[wid_TT+wid_fit-1:0]),
              .b(XLXN_52[wid_TT+wid_fit-1:0]),
              .aprime(XLXN_53[wid_TT+wid_fit-1:0]),
              .bprime(XLXN_54[wid_TT+wid_fit-1:0]));

    defparam XLXI_5.wid_TT=wid_TT;
    defparam XLXI_5.wid_fit=wid_fit;
sort2 XLXI_5 (.a(XLXN_55[wid_TT+wid_fit-1:0]),
              .b(XLXN_56[wid_TT+wid_fit-1:0]),
              .aprime(XLXN_57[wid_TT+wid_fit-1:0]),
              .bprime(XLXN_58[wid_TT+wid_fit-1:0]));

    defparam XLXI_6.wid_TT=wid_TT;
    defparam XLXI_6.wid_fit=wid_fit;
sort2 XLXI_6 (.a(XLXN_59[wid_TT+wid_fit-1:0]),
              .b(XLXN_60[wid_TT+wid_fit-1:0]),
              .aprime(XLXN_61[wid_TT+wid_fit-1:0]),
              .bprime(XLXN_62[wid_TT+wid_fit-1:0]));

    defparam XLXI_7.wid_TT=wid_TT;
    defparam XLXI_7.wid_fit=wid_fit;
sort2 XLXI_7 (.a(XLXN_63[wid_TT+wid_fit-1:0]),
              .b(XLXN_64[wid_TT+wid_fit-1:0]),
              .aprime(XLXN_65[wid_TT+wid_fit-1:0]),
              .bprime(XLXN_66[wid_TT+wid_fit-1:0]));

    defparam XLXI_8.wid_TT=wid_TT;
    defparam XLXI_8.wid_fit=wid_fit;
sort2 XLXI_8 (.a(XLXN_67[wid_TT+wid_fit-1:0]),
              .b(XLXN_68[wid_TT+wid_fit-1:0]),
              .aprime(XLXN_69[wid_TT+wid_fit-1:0]),
              .bprime(XLXN_70[wid_TT+wid_fit-1:0]));

    defparam XLXI_9.wid_TT=wid_TT;
    defparam XLXI_9.wid_fit=wid_fit;
sort2 XLXI_9 (.a(XLXN_71[wid_TT+wid_fit-1:0]),
              .b(XLXN_72[wid_TT+wid_fit-1:0]),
              .aprime(XLXN_73[wid_TT+wid_fit-1:0]),
              .bprime(XLXN_74[wid_TT+wid_fit-1:0]));
```

```
     defparam XLXI_10.wid_TT=wid_TT;
     defparam XLXI_10.wid_fit=wid_fit;
sort2 XLXI_10 (.a(XLXN_75[wid_TT+wid_fit-1:0]),
               .b(XLXN_76[wid_TT+wid_fit-1:0]),
               .aprime(XLXN_77[wid_TT+wid_fit-1:0]),
               .bprime(XLXN_78[wid_TT+wid_fit-1:0]));

     defparam XLXI_27.n=wid_TT+wid_fit;
 dff_N XLXI_27 (.CLK(CLK),
               .d(XLXN_49[wid_TT+wid_fit-1:0]),
               .q(XLXN_452[wid_TT+wid_fit-1:0]));

     defparam XLXI_28.n=wid_TT+wid_fit;
 dff_N XLXI_28 (.CLK(CLK),
               .d(XLXN_53[wid_TT+wid_fit-1:0]),
               .q(XLXN_453[wid_TT+wid_fit-1:0]));

     defparam XLXI_29.n=wid_TT+wid_fit;
 dff_N XLXI_29 (.CLK(CLK),
               .d(XLXN_57[wid_TT+wid_fit-1:0]),
               .q(XLXN_454[wid_TT+wid_fit-1:0]));

     defparam XLXI_30.n=wid_TT+wid_fit;
 dff_N XLXI_30 (.CLK(CLK),
               .d(XLXN_61[wid_TT+wid_fit-1:0]),
               .q(XLXN_455[wid_TT+wid_fit-1:0]));

     defparam XLXI_31.n=wid_TT+wid_fit;
 dff_N XLXI_31 (.CLK(CLK),
               .d(XLXN_65[wid_TT+wid_fit-1:0]),
               .q(XLXN_460[wid_TT+wid_fit-1:0]));

     defparam XLXI_32.n=wid_TT+wid_fit;
 dff_N XLXI_32 (.CLK(CLK),
               .d(XLXN_69[wid_TT+wid_fit-1:0]),
               .q(XLXN_464[wid_TT+wid_fit-1:0]));

     defparam XLXI_33.n=wid_TT+wid_fit;
 dff_N XLXI_33 (.CLK(CLK),
               .d(XLXN_73[wid_TT+wid_fit-1:0]),
               .q(XLXN_468[wid_TT+wid_fit-1:0]));
```

```
    defparam XLXI_34.n=wid_TT+wid_fit;
dff_N XLXI_34 (.CLK(CLK),
               .d(XLXN_77[wid_TT+wid_fit-1:0]),
               .q(XLXN_472[wid_TT+wid_fit-1:0]));

    defparam XLXI_35.n=wid_TT+wid_fit;
dff_N XLXI_35 (.CLK(CLK),
               .d(XLXN_50[wid_TT+wid_fit-1:0]),
               .q(XLXN_462[wid_TT+wid_fit-1:0]));

    defparam XLXI_36.n=wid_TT+wid_fit;
dff_N XLXI_36 (.CLK(CLK),
               .d(XLXN_54[wid_TT+wid_fit-1:0]),
               .q(XLXN_466[wid_TT+wid_fit-1:0]));

    defparam XLXI_37.n=wid_TT+wid_fit;
dff_N XLXI_37 (.CLK(CLK),
               .d(XLXN_58[wid_TT+wid_fit-1:0]),
               .q(XLXN_470[wid_TT+wid_fit-1:0]));

    defparam XLXI_38.n=wid_TT+wid_fit;
dff_N XLXI_38 (.CLK(CLK),
               .d(XLXN_62[wid_TT+wid_fit-1:0]),
               .q(XLXN_473[wid_TT+wid_fit-1:0]));

    defparam XLXI_43.n=wid_TT+wid_fit;
dff_N XLXI_43 (.CLK(CLK),
               .d(XLXN_66[wid_TT+wid_fit-1:0]),
               .q(XLXN_459[wid_TT+wid_fit-1:0]));

    defparam XLXI_44.n=wid_TT+wid_fit;
dff_N XLXI_44 (.CLK(CLK),
               .d(XLXN_70[wid_TT+wid_fit-1:0]),
               .q(XLXN_458[wid_TT+wid_fit-1:0]));

    defparam XLXI_45.n=wid_TT+wid_fit;
dff_N XLXI_45 (.CLK(CLK),
               .d(XLXN_74[wid_TT+wid_fit-1:0]),
               .q(XLXN_457[wid_TT+wid_fit-1:0]));

    defparam XLXI_46.n=wid_TT+wid_fit;
```

```
dff_N XLXI_46 (.CLK(CLK),
              .d(XLXN_78[wid_TT+wid_fit-1:0]),
              .q(XLXN_456[wid_TT+wid_fit-1:0]));

    defparam XLXI_55.n=wid_TT+wid_fit;
dff_N XLXI_55 (.CLK(CLK),
              .d(XLXN_452[wid_TT+wid_fit-1:0]),
              .q(XLXN_476[wid_TT+wid_fit-1:0]));

    defparam XLXI_56.n=wid_TT+wid_fit;
dff_N XLXI_56 (.CLK(CLK),
              .d(XLXN_453[wid_TT+wid_fit-1:0]),
              .q(XLXN_477[wid_TT+wid_fit-1:0]));

    defparam XLXI_57.n=wid_TT+wid_fit;
dff_N XLXI_57 (.CLK(CLK),
              .d(XLXN_454[wid_TT+wid_fit-1:0]),
              .q(XLXN_480[wid_TT+wid_fit-1:0]));

    defparam XLXI_58.n=wid_TT+wid_fit;
dff_N XLXI_58 (.CLK(CLK),
              .d(XLXN_455[wid_TT+wid_fit-1:0]),
              .q(XLXN_492[wid_TT+wid_fit-1:0]));

    defparam XLXI_59.n=wid_TT+wid_fit;
dff_N XLXI_59 (.CLK(CLK),
              .d(XLXN_461[wid_TT+wid_fit-1:0]),
              .q(XLXN_482[wid_TT+wid_fit-1:0]));

    defparam XLXI_60.n=wid_TT+wid_fit;
dff_N XLXI_60 (.CLK(CLK),
              .d(XLXN_465[wid_TT+wid_fit-1:0]),
              .q(XLXN_493[wid_TT+wid_fit-1:0]));

    defparam XLXI_61.n=wid_TT+wid_fit;
dff_N XLXI_61 (.CLK(CLK),
              .d(XLXN_469[wid_TT+wid_fit-1:0]),
              .q(XLXN_496[wid_TT+wid_fit-1:0]));

    defparam XLXI_62.n=wid_TT+wid_fit;
dff_N XLXI_62 (.CLK(CLK),
              .d(XLXN_475[wid_TT+wid_fit-1:0]),
```

```
              .q(XLXN_502[wid_TT+wid_fit-1:0]));

    defparam XLXI_63.n=wid_TT+wid_fit;
dff_N XLXI_63 (.CLK(CLK),
              .d(XLXN_463[wid_TT+wid_fit-1:0]),
              .q(XLXN_497[wid_TT+wid_fit-1:0]));

    defparam XLXI_64.n=wid_TT+wid_fit;
dff_N XLXI_64 (.CLK(CLK),
              .d(XLXN_467[wid_TT+wid_fit-1:0]),
              .q(XLXN_503[wid_TT+wid_fit-1:0]));

    defparam XLXI_65.n=wid_TT+wid_fit;
dff_N XLXI_65 (.CLK(CLK),
              .d(XLXN_471[wid_TT+wid_fit-1:0]),
              .q(XLXN_491[wid_TT+wid_fit-1:0]));

    defparam XLXI_66.n=wid_TT+wid_fit;
dff_N XLXI_66 (.CLK(CLK),
              .d(XLXN_474[wid_TT+wid_fit-1:0]),
              .q(XLXN_487[wid_TT+wid_fit-1:0]));

    defparam XLXI_67.n=wid_TT+wid_fit;
dff_N XLXI_67 (.CLK(CLK),
              .d(XLXN_459[wid_TT+wid_fit-1:0]),
              .q(XLXN_488[wid_TT+wid_fit-1:0]));

    defparam XLXI_68.n=wid_TT+wid_fit;
dff_N XLXI_68 (.CLK(CLK),
              .d(XLXN_458[wid_TT+wid_fit-1:0]),
              .q(XLXN_484[wid_TT+wid_fit-1:0]));

    defparam XLXI_69.n=wid_TT+wid_fit;
dff_N XLXI_69 (.CLK(CLK),
              .d(XLXN_457[wid_TT+wid_fit-1:0]),
              .q(XLXN_478[wid_TT+wid_fit-1:0]));

    defparam XLXI_70.n=wid_TT+wid_fit;
dff_N XLXI_70 (.CLK(CLK),
              .d(XLXN_456[wid_TT+wid_fit-1:0]),
              .q(XLXN_479[wid_TT+wid_fit-1:0]));
```

```
    defparam XLXI_145.n=wid_TT+wid_fit;
dff_N XLXI_145 (.CLK(CLK),
               .d(XLXN_476[wid_TT+wid_fit-1:0]),
               .q(XLXN_278[wid_TT+wid_fit-1:0]));

    defparam XLXI_146.n=wid_TT+wid_fit;
dff_N XLXI_146 (.CLK(CLK),
               .d(XLXN_477[wid_TT+wid_fit-1:0]),
               .q(XLXN_515[wid_TT+wid_fit-1:0]));

    defparam XLXI_147.n=wid_TT+wid_fit;
dff_N XLXI_147 (.CLK(CLK),
               .d(XLXN_481[wid_TT+wid_fit-1:0]),
               .q(XLXN_517[wid_TT+wid_fit-1:0]));

    defparam XLXI_148.n=wid_TT+wid_fit;
dff_N XLXI_148 (.CLK(CLK),
               .d(XLXN_494[wid_TT+wid_fit-1:0]),
               .q(XLXN_520[wid_TT+wid_fit-1:0]));

    defparam XLXI_149.n=wid_TT+wid_fit;
dff_N XLXI_149 (.CLK(CLK),
               .d(XLXN_483[wid_TT+wid_fit-1:0]),
               .q(XLXN_521[wid_TT+wid_fit-1:0]));

    defparam XLXI_150.n=wid_TT+wid_fit;
dff_N XLXI_150 (.CLK(CLK),
               .d(XLXN_498[wid_TT+wid_fit-1:0]),
               .q(XLXN_525[wid_TT+wid_fit-1:0]));

    defparam XLXI_151.n=wid_TT+wid_fit;
dff_N XLXI_151 (.CLK(CLK),
               .d(XLXN_495[wid_TT+wid_fit-1:0]),
               .q(XLXN_526[wid_TT+wid_fit-1:0]));

    defparam XLXI_152.n=wid_TT+wid_fit;
dff_N XLXI_152 (.CLK(CLK),
               .d(XLXN_500[wid_TT+wid_fit-1:0]),
               .q(XLXN_529[wid_TT+wid_fit-1:0]));

    defparam XLXI_153.n=wid_TT+wid_fit;
dff_N XLXI_153 (.CLK(CLK),
```

```
                .d(XLXN_499[wid_TT+wid_fit-1:0]),
                .q(XLXN_530[wid_TT+wid_fit-1:0]));

    defparam XLXI_154.n=wid_TT+wid_fit;
dff_N XLXI_154 (.CLK(CLK),
                .d(XLXN_501[wid_TT+wid_fit-1:0]),
                .q(XLXN_533[wid_TT+wid_fit-1:0]));

    defparam XLXI_155.n=wid_TT+wid_fit;
dff_N XLXI_155 (.CLK(CLK),
                .d(XLXN_490[wid_TT+wid_fit-1:0]),
                .q(XLXN_534[wid_TT+wid_fit-1:0]));

    defparam XLXI_156.n=wid_TT+wid_fit;
dff_N XLXI_156 (.CLK(CLK),
                .d(XLXN_486[wid_TT+wid_fit-1:0]),
                .q(XLXN_537[wid_TT+wid_fit-1:0]));

    defparam XLXI_157.n=wid_TT+wid_fit;
dff_N XLXI_157 (.CLK(CLK),
                .d(XLXN_489[wid_TT+wid_fit-1:0]),
                .q(XLXN_538[wid_TT+wid_fit-1:0]));

    defparam XLXI_158.n=wid_TT+wid_fit;
dff_N XLXI_158 (.CLK(CLK),
                .d(XLXN_485[wid_TT+wid_fit-1:0]),
                .q(XLXN_544[wid_TT+wid_fit-1:0]));

    defparam XLXI_159.n=wid_TT+wid_fit;
dff_N XLXI_159 (.CLK(CLK),
                .d(XLXN_478[wid_TT+wid_fit-1:0]),
                .q(XLXN_543[wid_TT+wid_fit-1:0]));

    defparam XLXI_160.n=wid_TT+wid_fit;
dff_N XLXI_160 (.CLK(CLK),
                .d(XLXN_479[wid_TT+wid_fit-1:0]),
                .q(XLXN_424[wid_TT+wid_fit-1:0]));

    defparam XLXI_191.n=wid_TT+wid_fit;
dff_N XLXI_191 (.CLK(CLK),
                .d(XLXN_278[wid_TT+wid_fit-1:0]),
                .q(aprime[wid_TT+wid_fit-1:0]));
```

```
    defparam XLXI_192.n=wid_TT+wid_fit;
dff_N XLXI_192 (.CLK(CLK),
               .d(XLXN_519[wid_TT+wid_fit-1:0]),
               .q(bprime[wid_TT+wid_fit-1:0]));

    defparam XLXI_193.n=wid_TT+wid_fit;
dff_N XLXI_193 (.CLK(CLK),
               .d(XLXN_518[wid_TT+wid_fit-1:0]),
               .q(cprime[wid_TT+wid_fit-1:0]));

    defparam XLXI_194.n=wid_TT+wid_fit;
dff_N XLXI_194 (.CLK(CLK),
               .d(XLXN_524[wid_TT+wid_fit-1:0]),
               .q(dprime[wid_TT+wid_fit-1:0]));

    defparam XLXI_195.n=wid_TT+wid_fit;
dff_N XLXI_195 (.CLK(CLK),
               .d(XLXN_522[wid_TT+wid_fit-1:0]),
               .q(eprime[wid_TT+wid_fit-1:0]));

    defparam XLXI_196.n=wid_TT+wid_fit;
dff_N XLXI_196 (.CLK(CLK),
               .d(XLXN_527[wid_TT+wid_fit-1:0]),
               .q(fprime[wid_TT+wid_fit-1:0]));

    defparam XLXI_197.n=wid_TT+wid_fit;
dff_N XLXI_197 (.CLK(CLK),
               .d(XLXN_528[wid_TT+wid_fit-1:0]),
               .q(gprime[wid_TT+wid_fit-1:0]));

    defparam XLXI_198.n=wid_TT+wid_fit;
dff_N XLXI_198 (.CLK(CLK),
               .d(XLXN_531[wid_TT+wid_fit-1:0]),
               .q(hprime[wid_TT+wid_fit-1:0]));

    defparam XLXI_199.n=wid_TT+wid_fit;
dff_N XLXI_199 (.CLK(CLK),
               .d(XLXN_532[wid_TT+wid_fit-1:0]),
               .q(iprime[wid_TT+wid_fit-1:0]));

    defparam XLXI_200.n=wid_TT+wid_fit;
```

```
dff_N XLXI_200 (.CLK(CLK),
               .d(XLXN_535[wid_TT+wid_fit-1:0]),
               .q(jprime[wid_TT+wid_fit-1:0]));

    defparam XLXI_201.n=wid_TT+wid_fit;
dff_N XLXI_201 (.CLK(CLK),
               .d(XLXN_536[wid_TT+wid_fit-1:0]),
               .q(kprime[wid_TT+wid_fit-1:0]));

    defparam XLXI_202.n=wid_TT+wid_fit;
dff_N XLXI_202 (.CLK(CLK),
               .d(XLXN_539[wid_TT+wid_fit-1:0]),
               .q(lprime[wid_TT+wid_fit-1:0]));

    defparam XLXI_203.n=wid_TT+wid_fit;
dff_N XLXI_203 (.CLK(CLK),
               .d(XLXN_540[wid_TT+wid_fit-1:0]),
               .q(mprime[wid_TT+wid_fit-1:0]));

    defparam XLXI_204.n=wid_TT+wid_fit;
dff_N XLXI_204 (.CLK(CLK),
               .d(XLXN_541[wid_TT+wid_fit-1:0]),
               .q(nprime[wid_TT+wid_fit-1:0]));

    defparam XLXI_205.n=wid_TT+wid_fit;
dff_N XLXI_205 (.CLK(CLK),
               .d(XLXN_542[wid_TT+wid_fit-1:0]),
               .q(oprime[wid_TT+wid_fit-1:0]));

    defparam XLXI_206.n=wid_TT+wid_fit;
dff_N XLXI_206 (.CLK(CLK),
               .d(XLXN_424[wid_TT+wid_fit-1:0]),
               .q(pprime[wid_TT+wid_fit-1:0]));

    defparam XLXI_282.wid_TT=wid_TT;
    defparam XLXI_282.wid_fit=wid_fit;
sort2 XLXI_282 (.a(XLXN_460[wid_TT+wid_fit-1:0]),
                .b(XLXN_462[wid_TT+wid_fit-1:0]),
                .aprime(XLXN_461[wid_TT+wid_fit-1:0]),
                .bprime(XLXN_463[wid_TT+wid_fit-1:0]));

    defparam XLXI_283.wid_TT=wid_TT;
```

```
    defparam XLXI_283.wid_fit=wid_fit;
sort2 XLXI_283 (.a(XLXN_464[wid_TT+wid_fit-1:0]),
                .b(XLXN_466[wid_TT+wid_fit-1:0]),
                .aprime(XLXN_465[wid_TT+wid_fit-1:0]),
                .bprime(XLXN_467[wid_TT+wid_fit-1:0]));

    defparam XLXI_284.wid_TT=wid_TT;
    defparam XLXI_284.wid_fit=wid_fit;
sort2 XLXI_284 (.a(XLXN_468[wid_TT+wid_fit-1:0]),
                .b(XLXN_470[wid_TT+wid_fit-1:0]),
                .aprime(XLXN_469[wid_TT+wid_fit-1:0]),
                .bprime(XLXN_471[wid_TT+wid_fit-1:0]));

    defparam XLXI_285.wid_TT=wid_TT;
    defparam XLXI_285.wid_fit=wid_fit;
sort2 XLXI_285 (.a(XLXN_472[wid_TT+wid_fit-1:0]),
                .b(XLXN_473[wid_TT+wid_fit-1:0]),
                .aprime(XLXN_475[wid_TT+wid_fit-1:0]),
                .bprime(XLXN_474[wid_TT+wid_fit-1:0]));

    defparam XLXI_286.wid_TT=wid_TT;
    defparam XLXI_286.wid_fit=wid_fit;
sort2 XLXI_286 (.a(XLXN_480[wid_TT+wid_fit-1:0]),
                .b(XLXN_482[wid_TT+wid_fit-1:0]),
                .aprime(XLXN_481[wid_TT+wid_fit-1:0]),
                .bprime(XLXN_483[wid_TT+wid_fit-1:0]));

    defparam XLXI_287.wid_TT=wid_TT;
    defparam XLXI_287.wid_fit=wid_fit;
sort2 XLXI_287 (.a(XLXN_492[wid_TT+wid_fit-1:0]),
                .b(XLXN_493[wid_TT+wid_fit-1:0]),
                .aprime(XLXN_494[wid_TT+wid_fit-1:0]),
                .bprime(XLXN_495[wid_TT+wid_fit-1:0]));

    defparam XLXI_288.wid_TT=wid_TT;
    defparam XLXI_288.wid_fit=wid_fit;
sort2 XLXI_288 (.a(XLXN_496[wid_TT+wid_fit-1:0]),
                .b(XLXN_497[wid_TT+wid_fit-1:0]),
                .aprime(XLXN_498[wid_TT+wid_fit-1:0]),
                .bprime(XLXN_499[wid_TT+wid_fit-1:0]));

    defparam XLXI_289.wid_TT=wid_TT;
```

```
    defparam XLXI_289.wid_fit=wid_fit;
sort2 XLXI_289 (.a(XLXN_502[wid_TT+wid_fit-1:0]),
                .b(XLXN_503[wid_TT+wid_fit-1:0]),
                .aprime(XLXN_500[wid_TT+wid_fit-1:0]),
                .bprime(XLXN_501[wid_TT+wid_fit-1:0]));

    defparam XLXI_290.wid_TT=wid_TT;
    defparam XLXI_290.wid_fit=wid_fit;
sort2 XLXI_290 (.a(XLXN_491[wid_TT+wid_fit-1:0]),
                .b(XLXN_488[wid_TT+wid_fit-1:0]),
                .aprime(XLXN_490[wid_TT+wid_fit-1:0]),
                .bprime(XLXN_489[wid_TT+wid_fit-1:0]));

    defparam XLXI_291.wid_TT=wid_TT;
    defparam XLXI_291.wid_fit=wid_fit;
sort2 XLXI_291 (.a(XLXN_487[wid_TT+wid_fit-1:0]),
                .b(XLXN_484[wid_TT+wid_fit-1:0]),
                .aprime(XLXN_486[wid_TT+wid_fit-1:0]),
                .bprime(XLXN_485[wid_TT+wid_fit-1:0]));

    defparam XLXI_298.wid_TT=wid_TT;
    defparam XLXI_298.wid_fit=wid_fit;
sort2 XLXI_298 (.a(XLXN_515[wid_TT+wid_fit-1:0]),
                .b(XLXN_517[wid_TT+wid_fit-1:0]),
                .aprime(XLXN_519[wid_TT+wid_fit-1:0]),
                .bprime(XLXN_518[wid_TT+wid_fit-1:0]));

    defparam XLXI_299.wid_TT=wid_TT;
    defparam XLXI_299.wid_fit=wid_fit;
sort2 XLXI_299 (.a(XLXN_520[wid_TT+wid_fit-1:0]),
                .b(XLXN_521[wid_TT+wid_fit-1:0]),
                .aprime(XLXN_524[wid_TT+wid_fit-1:0]),
                .bprime(XLXN_522[wid_TT+wid_fit-1:0]));

    defparam XLXI_300.wid_TT=wid_TT;
    defparam XLXI_300.wid_fit=wid_fit;
sort2 XLXI_300 (.a(XLXN_525[wid_TT+wid_fit-1:0]),
                .b(XLXN_526[wid_TT+wid_fit-1:0]),
                .aprime(XLXN_527[wid_TT+wid_fit-1:0]),
                .bprime(XLXN_528[wid_TT+wid_fit-1:0]));

    defparam XLXI_301.wid_TT=wid_TT;
```

```
            defparam XLXI_301.wid_fit=wid_fit;
        sort2 XLXI_301 (.a(XLXN_529[wid_TT+wid_fit-1:0]),
                        .b(XLXN_530[wid_TT+wid_fit-1:0]),
                        .aprime(XLXN_531[wid_TT+wid_fit-1:0]),
                        .bprime(XLXN_532[wid_TT+wid_fit-1:0]));

            defparam XLXI_302.wid_TT=wid_TT;
            defparam XLXI_302.wid_fit=wid_fit;
        sort2 XLXI_302 (.a(XLXN_533[wid_TT+wid_fit-1:0]),
                        .b(XLXN_534[wid_TT+wid_fit-1:0]),
                        .aprime(XLXN_535[wid_TT+wid_fit-1:0]),
                        .bprime(XLXN_536[wid_TT+wid_fit-1:0]));

            defparam XLXI_303.wid_TT=wid_TT;
            defparam XLXI_303.wid_fit=wid_fit;
        sort2 XLXI_303 (.a(XLXN_537[wid_TT+wid_fit-1:0]),
                        .b(XLXN_538[wid_TT+wid_fit-1:0]),
                        .aprime(XLXN_539[wid_TT+wid_fit-1:0]),
                        .bprime(XLXN_540[wid_TT+wid_fit-1:0]));

            defparam XLXI_304.wid_TT=wid_TT;
            defparam XLXI_304.wid_fit=wid_fit;
        sort2 XLXI_304 (.a(XLXN_544[wid_TT+wid_fit-1:0]),
                        .b(XLXN_543[wid_TT+wid_fit-1:0]),
                        .aprime(XLXN_541[wid_TT+wid_fit-1:0]),
                        .bprime(XLXN_542[wid_TT+wid_fit-1:0]));
    endmodule



    module ga(rnd0, rnd1, rnd2, rnd3, rnd4, rnd5, rnd6, rnd7, rnd8, rnd9, rnd10,
rnd11, rnd12, rnd13, rnd14, rnd15, min_fit, crc_sigs, crosscode, CLR, VALID, ITER, CLK,
w_out0, w_out1, w_out2, w_out3, w_out4, w_out5, w_out6, w_out7, w_out8, w_out9, w_out10,
w_out11, w_out12, w_out13, w_out14, w_out15);
            parameter wid_TT=14;
            parameter wid_fit=8;


            input [13:0] rnd0, rnd1, rnd2, rnd3, rnd4, rnd5, rnd6, rnd7, rnd8, rnd9,
rnd10, rnd11, rnd12, rnd13, rnd14, rnd15;
            input [63:0] crc_sigs;
            input [13:0] crosscode;
            input [7:0] min_fit;
            input CLR, VALID, ITER;
            input CLK /* synthesis syn_noclockbuf=1 syn_maxfan=100000 */ ;
```

```
output [13:0] w_out0, w_out1, w_out2, w_out3, w_out4, w_out5, w_out6,
w_out7, w_out8, w_out9, w_out10, w_out11, w_out12, w_out13, w_out14, w_out15;


wire [31:0] crc_adr;


wire [31:0] wcrc_adr;


wire [13:0] w_out0, w_out1, w_out2, w_out3, w_out4, w_out5, w_out6,
w_out7, w_out8, w_out9, w_out10, w_out11, w_out12, w_out13, w_out14, w_out15;


wire [wid_TT+wid_fit-1:0] w_new0, w_new1, w_new2, w_new3, w_new4, w_new5,
w_new6, w_new7, w_new8, w_new9, w_new10, w_new11, w_new12, w_new13, w_new14, w_new15;

wire [wid_TT+wid_fit-1:0] w_sort0, w_sort1, w_sort2, w_sort3, w_sort4,
w_sort5, w_sort6, w_sort7, w_sort8, w_sort9, w_sort10, w_sort11, w_sort12, w_sort13,
w_sort14, w_sort15;


//crc_sigs has the format of 32:1 CRC input value, 0 control line

//crc adr has the format of 8:1 adr lines, 0 control line

//If the control line is 1, the ROMs select the CRC address

CRC_calc crc(crc_sigs[32:1], wcrc_adr);


defparam ff_crc.n=32;

dff_eNB ff_crc(wcrc_adr, VALID&ITER, CLK, crc_adr);


defparam ff_crc_ctrl.n=1;

dff_eNB ff_crc_ctrl(crc_sigs[0], VALID&ITER, CLK, wcrc_ctrl);


assign t1 = w_new0;

assign t2 = min_fit;

//strgen(prev0, prev1, prev2, prev3, prev4, prev5, prev6, prev7, prev8,
prev9, prev10, prev11, prev12, prev13, prev14, prev15, reset_val,

//                    CLR, VALID, ITER, CLK,

//                    r_next0,   r_next1,   r_next2,   r_next3,   r_next4,
r_next5,  r_next6,  r_next7,  r_next8,  r_next9,  r_next10,r_next11,  r_next12,  r_next13,
r_next14, r_next15);

strgen sg(rnd0, rnd1, rnd2, rnd3, rnd4, rnd5, rnd6, rnd7, rnd8, rnd9,
rnd10, rnd11, rnd12, rnd13, rnd14, rnd15,

                                  w_out0[13:0],       w_out1[13:0],       w_out2[13:0],
w_out3[13:0],   w_out4[13:0],   w_out5[13:0],   w_out6[13:0],   w_out7[13:0],   w_out8[13:0],
w_out9[13:0],  w_out10[13:0],  w_out11[13:0],  w_out12[13:0],  w_out13[13:0],  w_out14[13:0],
w_out15[13:0],

                              min_fit, CLR, VALID, ITER, CLK,

                              w_new0,  w_new1,  w_new2,  w_new3,  w_new4,  w_new5,  w_new6,
w_new7, w_new8, w_new9, w_new10, w_new11, w_new12, w_new13, w_new14, w_new15);


defparam s.wid_TT=wid_TT;

defparam s.wid_fit=wid_fit;
```

```
                sort s(w_new0, w_new1, w_new2, w_new3, w_new4, w_new5, w_new6, w_new7,
w_new8, w_new9, w_new10, w_new11, w_new12, w_new13, w_new14, w_new15, CLK,

                                w_sort0, w_sort1, w_sort2, w_sort3, w_sort4, w_sort5,
w_sort6, w_sort7, w_sort8, w_sort9, w_sort10, w_sort11, w_sort12, w_sort13, w_sort14,
w_sort15);


                defparam c.wid_TT=wid_TT;

                defparam c.wid_fit=wid_fit;

                crossmut c(w_sort0, w_sort1, w_sort2, w_sort3, w_sort4,

                                        w_sort5, w_sort6, w_sort7, w_sort8, w_sort9,

                                        w_sort10, w_sort11, w_sort12, w_sort13,

                                        w_sort14, w_sort15,

                                        {crc_adr[7:0], wcrc_ctrl}, crosscode, CLR, VALID,
ITER, CLK,

                                        w_out0[wid_TT-1:0],             w_out1[wid_TT-1:0],
w_out2[wid_TT-1:0],     w_out3[wid_TT-1:0],     w_out4[wid_TT-1:0],     w_out5[wid_TT-1:0],
w_out6[wid_TT-1:0], w_out7[wid_TT-1:0],

                                        w_out8[wid_TT-1:0],             w_out9[wid_TT-1:0],
w_out10[wid_TT-1:0],    w_out11[wid_TT-1:0],    w_out12[wid_TT-1:0],    w_out13[wid_TT-1:0],
w_out14[wid_TT-1:0], w_out15[wid_TT-1:0]);


        endmodule
```

LIST OF REFERENCES

[1] K. E. Batcher, "Sorting networks and their applications," *Spring Joint Computer Conference*, pp. 307–314, 1968.

[2] J. T. Butler, G. W. Dueck, S. N. Yanushkevich and V. P. Shmerko Discrete, "On the number of generators for transeunt triangles," *Applied Mathematics 108*, pp. 309–316, 2001.

[3] J. T. Butler and T. Sasao, "Logic functions for cryptography—A Tutorial," *Proceedings of the Reed-Muller Workshop 2009*, pp. 127–136, Naha, Okinawa, Japan, May 23–24, 2009.

[4] D. A. Coley, *An Introduction to Genetic Algorithms for Scientists and Engineers*, World Scientific Publishing Co. Pte. Ltd., River Edge, New Jersey, 1999.

[5] T.W. Cusick and P. Stanica, *Cryptographic Boolean Functions and Applications*, Academic Press, San Diego, California, 2009.

[6] J. F. Dillon "A Survey of Bent Functions," *NSA Technical Journal Special Issue*, pp. 191–215, 1972.

[7] M. Hell, T. Johansson, A. Maximov,W. Meier, "The Grain family of stream ciphers**,"** *New Stream Cipher Designs - The eSTREAM Finalists*, pp. 179–190, Springer-Verlag, 2008.

[8] P. Langevin, "Classification of Boolean Quartics Forms in eight Variables," http://langevin.univ-tln.fr/project/quartics/, last accessed 31AUG09.

[9] M. Mitchell, *An Introduction to Genetic Algorithms*, The MIT Press, Cambridge, Massachusetts, 1996.

[10] W. Meier and O. Staffelbach, "Nonlinearity criteria for cryptographic functions," *Advances in Cryptology, Proc. Eurocrypt'89, LNCS 434*, pp. 549–562, Springer-Verlag, 1990.

[11] A. Perez, "Byte-wise CRC Calculations," *IEEE Micro, vol. 3, no. 3*, pp. 40–50, 1983.

[12] B. Preneel, "Analysis and design of cryptographic has functions," PhD. Thesis, Katholieke Universiteit, Leuven, Belgium, 1993.

[13] O.S. Rothaus, "On "bent" functions," *J. Combin. Theory (A) 20*, pp. 300–305, 1976.

[14] C. Shannon, “Communication Theory of Secrecy Systems,” *Bell System Technical Journal, vol.28(4),* pp. 656–715, 1949.

[15] SRC Computers, Inc., “SRC Carte™ C Programming Environment v2.2 Guide,” SRC–007–18, Colorado Springs, Colorado, August 2006.

[16] SRC Computers, Inc., “SRC Training Course,” Colorado, Springs, Colorado, 2005.

[17] R. Sung, A. Sung, P. Chan, J. Mah, “Linear Feedback Shift Register” http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/1999f/Drivers_Ed/lfsr.html, last accessed 31AUG09.

[18] V.P. Suprun, “Fixed polarity Reed-Muller expressions of symmetric Boolean functions,” *Proceedings of IFIP WG 10.5 Workshop on Application of the Reed-Muller Expansions in Circuit Design*, pp. 246–249, 1995.

[19] M. Vavouras, K. Papadimitriou and I. Papaefstathiou, “High-speed fpga-based implementations of a geneticalgorithm,” *Proceedings of the IEEE International Symposium on Systems, Architectures, Modeling and Simulation*, Samos, Greece, July 2009.

[20] R. N. Williams, “A Painless Guide to CRC Error Detection Algorithms,” http://www.cs.waikato.ac.nz/~312/crc.txt, last accessed 31AUG09.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, VA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

3. Dr. John G. Harkins
   National Security Agency
   Fort Meade, MD

4. Dr. David R. Podany
   National Security Agency
   Fort Meade, MD

5. Mr. David Caliga
   SRC Computers
   Colorado Springs, CO

6. Mr. Jon Huppenthal
   SRC Computers
   Colorado Springs, CO

7. Dr. Jeff Hammes
   SRC Computers
   Colorado Springs, CO

8. Dr. Thad Welch
   Electrical and Computer Engineering Department
   Boise State University
   Boise, ID

9. Dr. Douglas Fouts
   Naval Postgraduate School
   Monterey, CA

10. Dr. Herschel Loomis
    Naval Postgraduate School
    Monterey, CA

11. Mr. Kyprianos Papadimitriou
ECE Dept.
Technical University of Crete

12. Dr. Bret Michael
Naval Postgraduate School
Monterey, CA

13. Dr. Ted Huffmire
Naval Postgraduate School
Monterey, CA

14. Dr. Jon T. Butler
Naval Postgraduate School
Monterey, CA

15. Dr. Pantelimon Stanica
Naval Postgraduate School
Monterey, CA

16. Dr. Sherif Michael
Naval Postgraduate School
Monterey, CA

17. Mr. Apostolos Dollas
ECE Dept.
Technical University of Crete