2010-12

An analysis of cryptographically significant Boolean functions with high correlation immunity by reconfigurable computer

Etherington, Carole J.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

AN ANALYSIS OF CRYPTOGRAPHICALLY SIGNIFICANT BOOLEAN FUNCTIONS WITH HIGH CORRELATION IMMUNITY BY RECONFIGURABLE COMPUTER

by

Carole J. Etherington

December 2010

Thesis Co-Advisors: Jon T. Butler
Pantelimon Stanica

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

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** December 2010	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE An Analysis of Cryptographically Significant Boolean Functions With High Correlation Immunity by Reconfigurable Computer		**5. FUNDING NUMBERS** N/A	
6. AUTHOR(S) Carole J Etherington			
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. IRB Protocol number: N/A.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited		**12b. DISTRIBUTION CODE**	

13. ABSTRACT (maximum 200 words)

Boolean functions with high correlation immunity can be used in cryptosystems to defend against correlation attacks. These functions are rare and difficult to find. As the variables increase, this task becomes exponentially more complex and time consuming. Three different ways to execute a program to find the correlation immunity of a function are compared in this thesis. First, a program was written in C and executed on a conventional CPU. The same program was then executed on an FPGA on the SRC-6 reconfigurable computer. A similar program was written in Verilog and executed on the FPGA. By taking advantage of the parallel processing ability of the SRC-6, a well-programmed Verilog macro can find functions with high correlation immunity at a much faster rate.

The SRC-6 reconfigurable computer is used in this thesis to find the correlation immunity of millions of functions up to six variables. Rotation symmetric and balanced functions were examined to find subsets that contain a high percentage of functions with good correlation immunity. The nonlinearity and correlation immunity of functions of four and five variables were compared to find functions with the best balance to fend off both correlation and linear attacks on a cryptosystem.

14. SUBJECT TERMS Correlation Immunity, Cryptology, Field Programmable Gate Array (FPGA), Reconfigurable Computer, Rotation Symmetric Functions, Bent Functions, Balanced Functions			**15. NUMBER OF PAGES** 130
			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

AN ANALYSIS OF CRYPTOGRAPHICALLY SIGNIFICANT BOOLEAN FUNCTIONS WITH HIGH CORRELATION IMMUNITY BY RECONFIGURABLE COMPUTER

Carole J. Etherington
Lieutenant, United States Navy
B.S., University of Kentucky, 2004

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 2010

Author: Carole J. Etherington

Approved by: Jon T. Butler
Thesis Co-Advisor

Pantelimon Stanica
Thesis Co-Advisor

Clark Robertson
Chairman, Department of Electrical Engineering and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Boolean functions with high correlation immunity can be used in cryptosystems to defend against correlation attacks. These functions are rare and difficult to find. As the number of variables increases, this task becomes exponentially more complex and time consuming. Three different ways to execute a program to find the correlation immunity of a function are compared in this thesis. First, a program was written in C and executed on a conventional CPU. The same program was then executed on an FPGA on the SRC-6 reconfigurable computer. A similar program was written in Verilog and executed on the FPGA. By taking advantage of the parallel processing ability of the SRC-6, a well-programmed Verilog macro can find functions with high correlation immunity at a much faster rate.

The SRC-6 reconfigurable computer is used in this thesis to find the correlation immunity of millions of functions of up to six variables. Rotation symmetric and balanced functions were examined to find subsets that contain a high percentage of functions with good correlation immunity. The nonlinearity and correlation immunity of functions of four and five variables were compared to find functions with the best balance to fend off both correlation and linear attacks on a cryptosystem.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Cryptographically significant functions with a concentration on functions with high correlation immunity are examined in this thesis. For the first time, the SRC-6 reconfigurable computer was used to find the correlation immunity of millions of Boolean functions up to six variables. Smaller subsets of Boolean functions thought to contain a higher percentage of functions with good correlation immunity were examined. These subsets included the set of rotation symmetric functions and the set of balanced functions. The results and analysis from these tests are discussed in this thesis. The SRC-6 was used to compute the correlation immunity and nonlinearity of Boolean functions up to five variables in order to find functions with the highest degree of both these properties. A comparison between the computation times of a Verilog program executed on an FPGA associated with the SRC-6 reconfigurable computer and a C program executed on the PC showed that the SCR-6 was able to generate and test functions at a much faster rate than the PC. An additional comparison showed the speed up achieved on the SRC-6 by using a well-programmed Verilog macro over a program written in C.

Boolean functions are of great importance when designing running key generators for stream ciphers in cryptosystems. These stream ciphers are responsible for encrypting binary digits of plaintext one digit at a time into ciphertext. The ability to defend a system against cryptanalysis depends on the Boolean function used for the combiner function. This function must meet certain criteria to yield a cryptographically secure scheme that can resist known attacks such as linear and correlation attacks. The ideal function would be bent, have high correlation immunity, and be balanced. Unfortunately, functions with all these qualities do not exist; therefore, a balance between these qualities must be found. This thesis is primarily focused on finding functions with high correlation immunity, but a comparison between the correlation immunity, linearity and balancedness is also examined and discussed.

The Boolean function chosen for the running key generator must have a high degree of correlation immunity to successfully defend a cryptosystem against a correlation attack. Correlation immunity was defined by T. Siegenthaler in 1983 in

response to correlation attacks. These attacks exploit a statistical weakness found in functions with low correlation immunity. Functions with high correlation immunity are uncommon and difficult to find. No practical method is known to create functions with high correlation immunity, so exhaustive searches are required. By using the SRC-6 reconfigurable computer, the time required to find these functions was greatly reduced.

In order to find the most effective way to generate and test functions for correlation immunity, three similar programs were written and the computation time was compared. The first program was written in the C programming language and executed on an Intel Xeon processor running at 2.8 GHz, which is one of the two microprocessors associated with the SRC-6. The C code was also modified to run on the SRC-6 reconfigurable computer, which operates at a clock speed of 100 MHz. A Verilog macro was written to find the correlation immunity of a function generated in C code. This macro was designed to take advantage of the parallel processing ability of the field programmable gate arrays (FPGA) and the capability of the system to pipeline a program. With these advantages, the SRC-6 was able to output the correlation immunity of a set of functions at a rate of one result per clock cycle. Using the Verilog macro for four variables on the SRC-6, we obtained a speed-up of close to 400 times the PC code and 1800 times over the C code run on SRC-6.

With the ability to test one function every clock cycle, 100,000,000 functions can be tested every second. This value is only limited by the clock speed of the SRC-6. Even at this rate, all functions with six variables cannot be exhaustively tested in a feasible amount of time. By finding smaller test sets and relationships between functions with other cryptographically significant properties, functions of larger number of variables with higher correlation immunity may be found.

LIST OF ACRONYMS AND ABBREVIATIONS

FPGA	Field Programmable Gate Array
FUT	Function Under Test
LFSR	Linear Feedback Shift Register
LUT	Lookup Table
MAP	Multi-Adaptive Processing
MUX	Multiplexor
TT	Truth Table

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to express my appreciation to Dr. Jon Butler for his endless time and support.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM DEFINITION

Functions with high correlation immunity are vital in defending cryptosystems against correlation attacks. These functions are rare, and the only known practical way to find these functions is through an exhaustive search. In order to test for the correlation immunity of a Boolean function, a program written in the C programming language was executed on both the SRC-6 reconfigurable computer with a 100 MHz clock and on the Intel Xeon processor that operates at 2.8 GHz. A similar program, written as a Verilog macro for computation and C code for enumeration, was compiled and run on the SRC-6. The execution times were compared to find the most effective way to generate and test the correlation immunity of millions of functions. The time required to evaluate all functions with more than five variables makes an exhaustive search of all functions infeasible. Examining the properties of functions with high correlation immunity helps in discovering subsets of Boolean functions rich in functions with high correlation immunity. This allows functions with high correlation immunity of more variables to be found faster with less testing. The correlation immunity of all rotation symmetric functions and balanced functions was examined in the search for smaller test sets. In addition, functions that possess other cryptographically significant properties, such as high nonlinearity, are extremely important to find. This way the function can defend against other known attacks. The relationship between nonlinearity and correlation immunity of all functions up to five variables was examined. The results were examined to find functions that had high degrees of multiple cryptographic properties.

B. BACKGROUND

The correlation immunity of a Boolean function was defined by T. Siegenthaler in 1983 in response to correlation attacks [8]. Correlation immunity is a measure of the degree the outputs of a Boolean function are uncorrelated with different subsets of the function's inputs. Boolean functions with low order correlation immunity are more

susceptible to correlation attacks than functions with higher order correlation immunity. Discovering effective techniques to find functions with high correlation immunity are of great value in the field of cryptology.

Cryptology is a widely used tool in communications, computer networks, and computer security. Some of its applications include ATM cards, computer passwords, remote computer login and commerce. Since World War II and the development of digital computers, the methods used for encrypting data have become increasingly more complex and the application of encryption more widespread. These advances have also led to more advanced cryptanalysis, the study of methods for decrypting information without the key, which has created the need for more complex ciphers. A cipher is a pair of algorithms that are used to encrypt and decrypt data. The stream cipher used in running key generators is made up of multiple linear feedback shift registers (LSFR) that are combined by a Boolean function to form the keystream. The keystream is then bitwise Exclusive-OR'd with the plaintext to create the ciphertext. A typical keystream generator for a cryptosystem is illustrated in Figure 1. The symbols $S_1,S_2,\ldots,S_n$ represent the LSFRs, f represents the combiner function, a Boolean function of n variables, and k is the keystream.

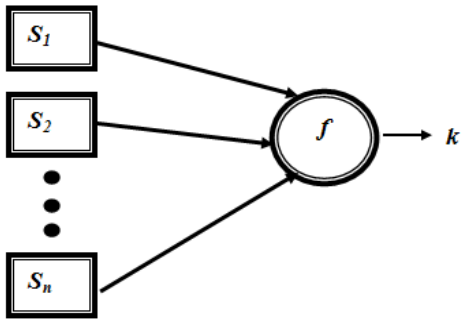

Figure 1. Keystream generator from [6].

Most of the reliability and security of the cryptosystems lies in the Boolean function used for the combiner function. Correlation attacks are possible when there is a

significant correlation between the output state of an individual LSFR in the keystream generator and the output of the combiner function. When this occurs, the attacker is able to recover the different initializations of the LSFRs separately to reduce the cost of an exhaustive search. This greatly reduces the complexity of the attack.

The complexity of a brute force attack on the system is $\prod_{i=1}^{n}\left(2^{L_i}-1\right)$ where L_i is the length of the *i*-th LSFR. If the combining function is chosen to create a relationship between the keystream and the output of the *i*-th LSFR, then it is possible to only try all $2^{L_i}-1$ possible initiations of the *i*-th LSFR; it is highly probable that the correct initiation will be detected. This reduces the complexity of a brute force attack to $\prod_{j=1, j\neq i}^{n}\left(2^{L_j}-1\right)+2^{L_i}-1$ [5]. If the combining function has correlation immunity *k*, then the correlation attack must consider at least $(k+1)$ different registers simultaneously.

While high correlation immunity is necessary when choosing the Boolean function for the keystream generator, it is not sufficient to make the cryptosystem secure. Most combiner functions combine criteria such as balancedness, nonlinearity, high correlation immunity and high algebraic immunity to ensure resistance to known attacks. Since functions that have the best of all these properties do not exist, necessary trade-offs must be considered when choosing the combiner function in a keystream generator. We first look at finding functions with high correlation immunity and then evaluate these functions for other desirable cryptographic features.

C. METHOD

The only known primary methods for constructing resilient functions are for small numbers of variables and do not allow for designing functions with a high degree or high nonlinearity [7]. Secondary constructions use previously defined functions to create new ones. These techniques use recursive algorithms that are not adequate in realistic applications. Since there is no known practical way to generate a function with high correlation immunity, an exhaustive search of all functions must be performed. To test a function of *n* variables for correlation immunity *k*, a function is broken into 2^k subsets *C(n,k)* times. Each time the number of ones in each subset is calculated and compared to

the number of ones in all other subsets. As n increases, so does the number of tests for each correlation immunity value and, therefore, the time required to fully test each function.

When using a conventional CPU, each function is tested for every value of correlation immunity before the next function is processed. This is very inefficient. The SRC-6 allows the use of a type of parallel programming called pipelining. This ability is extremely valuable when a program has a long delay and can be split into multiple steps to reach the final result. When finding the correlation immunity of a function, the incoming function under test (FUT) does not depend on the results of the previous FUT and, thus, allows pipelining. Here, the incoming function can start the first step as soon as the previous function moves to the next step in the testing. By breaking the program into efficient steps, once the first function completes the final phase an output will result every clock cycle. The total run time then depends on the clock speed. The SRC-6 reconfigurable computer used for this thesis had a clock speed of 100 MHz, which allowed 100,000,000 functions to be tested every second.

D. THESIS OUTLINE

The introduction, including the objective and background information, is contained in Chapter I. Definitions and lemmas used in this thesis and a discussion on finding functions with high correlation immunity are discussed in Chapter II. The circuit used in this thesis is discussed and examined in Chapter III. The analysis of the results is contained in Chapter IV. The conclusion and recommendations for further work are contained in Chapter V. The code for the SRC-6 is contained in Appendix A. The code for the PC is contained in Appendix B.

II. CRYPTOGRAPHICALLY SIGNIFICANT FUNCTIONS

A. BACKGROUND

1. Definitions

a. Boolean Functions

A Boolean function f on n variables is a map from the n-dimensional vector space V_n to F_2, the two-element field. For a function f, let $f_0 = f(0,0,...,0)$, $f_1 = f(0,0,...,1)$, ..., and $f_{2^n-1} = f(1,1,...,1)$. The sequence of bits, TT= (f_0 f_1 ... f_{2^n-1}) is the truth table representation of f [2].

b. Correlation Immunity

An n-variable function f has correlation immunity of order k if and only if, for every fixed set of S of k variables, $1 \le k \le n$, and for every assignment of values to the variables in S, the weights of all subfunctions are the same [1].

Example: The following truth table shows the output for the function $f(x_1, x_2, x_3, x_4) = x_1 \oplus x_2 \oplus x_3 \oplus x_4$. To test this function for correlation immunity of 2, all possible combinations of two variables must be considered. There are six ways to choose two out of the four variables. For this example, x_1 and x_2 are the two selected variables. These two variables can be assigned four different values, 00, 01, 10 and 11. In order to pass this test for correlation immunity, the number of ones in the function value for each subset of these two variables must be the same. For this function to have correlation immunity of 2, all combinations of two variables must pass this same test. The number of ones for each possible value of x_1 and x_2 is shown in Table 1. In this case, the number of ones are the same so the test is passed.

Table 1. The results of one test for a correlation immunity of two [11].

$x_1x_2x_3x_4$	f	00	01	10	11
0000	0	0			
0001	1	1			
0010	1	1			
0011	0	0			
0100	1		1		
0101	0		0		
0110	0		0		
0111	1		1		
1000	1			1	
1001	0			0	
1010	0			0	
1011	1			1	
1100	0				0
1101	1				1
1110	1				1
1111	0				0
# of ones	8	2	2	2	2

c. Linear Functions

A linear function is the Exclusive-OR of one or more variables or the constant zero function [2].

Example: $f(x_1, x_2, x_3) = x_1 \oplus x_2$

d. Affine Functions

An affine function is a linear function or the complement of a linear function [2].

Example: $f(x_1, x_2, x_3) = x_1 \oplus x_2 \oplus 1$

e. Hamming Distance

The Hamming distance between two functions is the number of places where their truth table representations disagree [2].

f. Nonlinearity

The nonlinearity of a function f is the minimum Hamming distance between f and all affine functions [2].

g. Bent Functions

A bent function has the largest nonlinearity of all Boolean functions [2].

h. Rotation Symmetric

A function is rotation symmetric if it is invariant under all cyclic rotations of the inputs. Rotationly symmetric functions can be divided into orbits so that each orbit consists of all cyclic shifts of one input [9].

Example: The truth table for the function $f = x_1 \oplus x_2 \oplus x_3 \oplus x_4$ is illustrated in Table 2. This function is a rotation symmetric function of four variables. One cycle is shown for each of the six orbits in the top row of the table. Since the value remains unchanged for all cyclic rotations of that orbit, the function is rotation symmetric.

Table 2. Rotation symmetric truth table.

$x_1x_2x_3x_4$	f	0000	0001	0011	0101	0111	1111
0000	0	0					
0001	1		1				
0010	1		1				
0011	0			0			
0100	1		1				
0101	0				0		
0110	0			0			
0111	1					1	
1000	1		1				
1001	0			0			
1010	0				0		
1011	1					1	
1100	0			0			
1101	1					1	
1110	1					1	
1111	0						0
f value		0	1	0	0	1	0

i. Balanced Functions

A balanced function has the same number of 1s and 0s in its truth table form.

j. Resiliency

A balanced function with correlation immunity *k* is said to be *k*-resilient [5].

2. Lemmas

Lemma 2.1. If a function f has correlation immunity of order k, then f is also correlation immune of k−1 and may or may not have correlation immunity of order k+1. [6]

Lemma 2.2. *The complement of a function has the same correlation immunity as that of the original function.*

Proof: By the definition of correlation immunity, an *n*-variable function *f* has correlation immunity of order *k* if and only if, for every fixed set of *S* of *k* variables, $1 \le k \le n$, and for every assignment of values to the variables in *S*, the weights of all subfunctions are the same. The weight is calculated by the number of 1s in each subfunction. If the size of each of the subfunctions is the same, then the number of 0s in each subfunction must be the same. This implies that the weight of every subfunction of *f*'s complement is the same. Therefore, the complement of a function has the same correlation immunity of the original function. Q.E.D.

Lemma 2.3. *The only functions with correlation immunity n are the zero functions and the function whose TT value consists of all ones.*

Proof: For a function to have correlation immunity of order *n*, all 2^n subsets must have an equal number of 1s. Since the truth table for a function of *n* variables only has 2^n values, then they must all be the same for all TT entries. The only way to have all values the same is either all ones or all zeros. Therefore, the only functions with correlation immunity *n* are the constant zero and one. Q.E.D

B. CRYPTOGRAPHICALLY SIGNIFICANT SUBSETS

1. Rotation Symmetric Functions

Rotation Symmetric functions are functions whose value remains unchanged when the variables in the function are rotated circularly to each of the possible positions. The total space of these functions is much smaller ($\approx 2^{2^n/n}$) than the space of all Boolean

functions (2^{2^n}). It has been experimentally shown that this set contains functions with very desirable cryptographically properties, such as good algebraic immunity (resistance to algebraic attack), balancedness, high correlation immunity and algebraic degree [9].

2. Balanced Functions

A balanced function has an equal number of 1s and 0s in its truth table values. Balancedness is an important cryptographic criteria for designing the combiner function in order to prevent the system from leaking statistical information on the plaintext when given the ciphertext. The number of balanced functions is $C\left(2^n, 2^{n-1}\right)$. Although this space is quite large, other properties of functions with high correlation immunity can be used to decrease the size. If the correlation immunity of a function is *k*, these balanced functions are *k*-resilient.

3. Nonlinearity

In order to break the linear property of the LSFRs, increase the period complexity of the output sequence and avoid linear attack, the combining function must be highly nonlinear. The nonlinearity of a function is the minimum Hamming distance between f and all affine functions. Functions with the highest nonlinearity are called bent functions. Bent functions are not balanced and therefore will have a correlation immunity of at most n−2. It has been shown that the nonlinearity of any k-resilient function is smaller than or equal to $2^{n-1} - 2^{k+1}$ if $n/2 - 1 < k + 1$, to $2^{n-1} - 2^{n/2-1} - 2^{k+1}$ if n is even and $n/2 - 1 \geq k + 1$ and to $2^{n-1} - 2^{k+1}[2^{n/2-k-2}]$ if n is odd and $n/2 - 1 \geq k + 1$ [7].

C. TESTING FOR CORRELATION IMMUNITY

For functions of five variables or less, all functions can be exhaustively tested in a reasonable amount of time. These functions were all tested on the SRC-6 reconfigurable computer. The code is included in Appendix A. The correlation immunity distribution for all functions of *n*=3, 4 and 5 variables is shown in Table 3.

Table 3. Distribution of correlation immunity for *n*=3,4 and 5.

n / Correlation Immunity	0	1	2	3	4	5
3	236	16	2	2	0	0
4	64,888	636	8	2	2	0
5	4,291,827,234	3,139,004	1044	10	2	2

The rare nature of functions with high correlation immunity is shown in Table 3. The number of functions with a given correlation immunity is known mathematically for functions of any number of variables, but how to construct those functions is not [14]. The functions with correlation immunity *n* are the zero function and the function whose TT values are all 1 which are not of interest for cryptology. As the number of variables increases, the percentage of functions with correlation immunity greater than 0 decreases exponentially. This makes finding these functions increasingly harder as the variable size increases.

A common size of a function used for cryptosystems is 1024 bits, which is a function of ten variables. Since most computers have a maximum register size of 64 bits, finding the correlation immunity of a function greater than five variables becomes much more complicated. The SRC-6 allows registers of any size to be created. The limitation for finding the correlation immunity of a function depends on the total number of functions and the clock speed of the FPGA. The clock speed is directly related to the number of functions that can be tested. Since the number of functions increases exponentially, the clock speed needs to be thousands of times faster just to feasibly test for a function of six variables. Increasing the clock speed is not alone sufficient to effectively test for the correlation immunity of functions of high variables. Therefore, the size of the test set needs to be reduced.

A program was written in MATLAB to find and graph the correlation immunity for all functions of four variables. The results are shown in Figure 2.

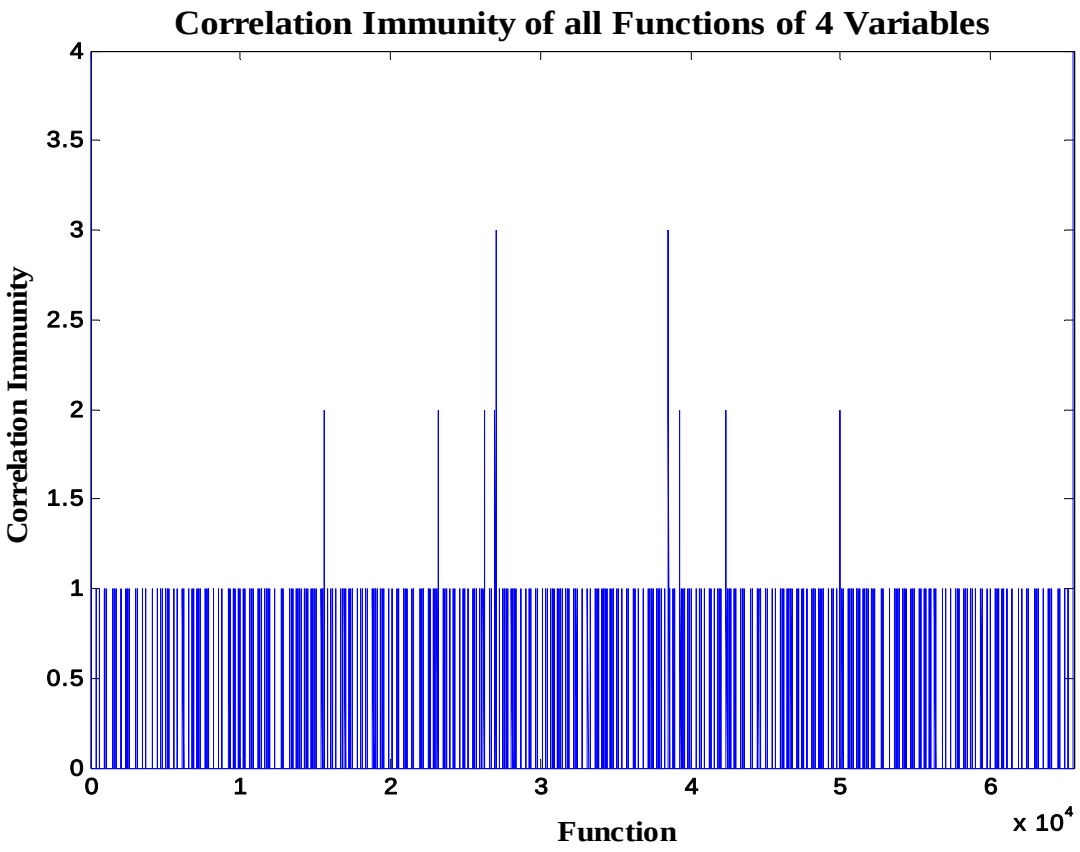

Figure 2. Distribution of correlation immunity for all functions, n=4.

The results for 65,536 different functions of four variables are shown in Figure 2. The program tested the binary representation for the values of 0 (0000 0000 0000 0000 in binary) through 65,535 (1111 1111 1111 1111 in binary). The zero function and the function whose truth table values are all 1 have correlation immunity of 4 but are not of interest for cryptographic purposes. Most of the functions have correlation immunity of 0, so the graph in Figure 2 emphasizes the functions with correlation immunity greater than 0. This is a very small percentage of all Boolean functions. The results are symmetric around the center of the test group. This supports Lemma 2.2 that states the correlation immunity of a function's complement is the same as the function's. By only testing the first half of all Boolean functions of a given variable, the number of functions needing to be tested can be cut in half.

Figure 4 is a close up of Figure 3. It can be seen that all funtions of value 255 (0000 0000 1111 1111) and less have correlation immunity 0 with the exception of the zero function.

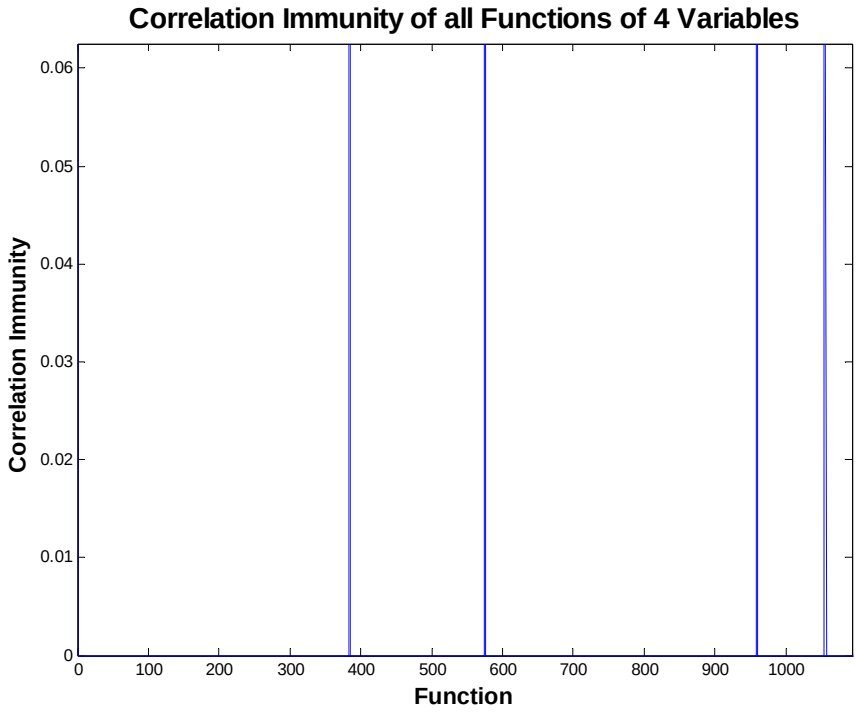

Figure 3. A close up section of the graph of correlation immunity of all functions, n=4.

This is due to the fact that a function that has a correlation immunity of 1 for a function of four variables requires that the function's TT values have an equal number of 1s for each subset of eight bits for the four required tests. One such test divides the function into a set of the first eight bits and a set of the second eight bits. Any function whose decimal value is less than or equal to 255 will not have a 1 in the first eight bits. Therefore, that function will fail at least one test for a correlation immunity of 1 which results in a correlation immunity of 0, with the exception of the zero function. The test size can be reduced by $2^{2^n/2}$. In fact, in the case of functions of four variables, the first function with correlation immunity of 1 does not occur until close to the functions whose decimal value is 400. By finding more trends, the test size can be further reduced.

D. SUMMARY

Definitions for terms used throughout this thesis which included descriptions of important characteristics for combiner functions used in cryptosystems were provided in this thesis. Testing for the correlation immunity was discussed. The implementation of the circuit used to find the correlation immunity of a function is covered in the next chapter.

III. IMPLEMENTATION

A. SRC-6 CIRCUIT

A circuit was created to compute the correlation immunity of a function of n variables. The block form is shown in Figure 4.

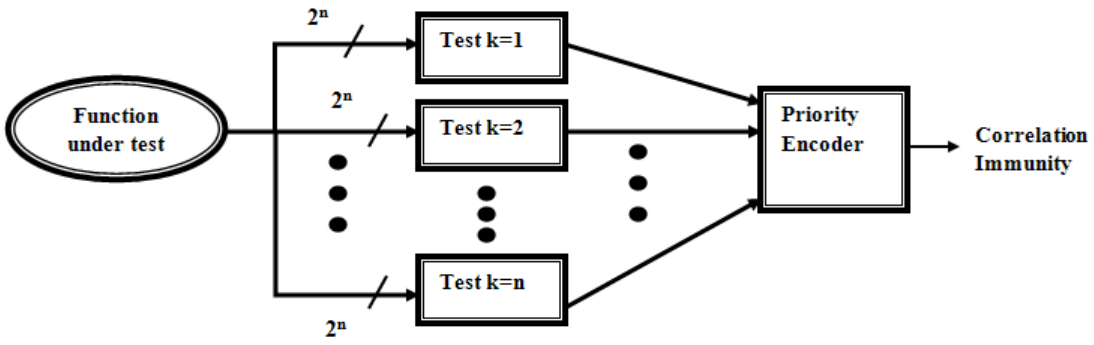

Figure 4. Circuit for computing the correlation immunity of a function.

This circuit was built using the Verilog programming language and executed on the SRC-6 reconfigurable computer. The SRC-6 uses the Xilinx Vertex2 Pro FPGA. The circuit takes a function of size 2^n and tests that function for correlation immunity 1 through n simultaneously. The priority encoder receives a one-bit signal from each test block and outputs the correlation immunity of the function. A more thorough look at the test blocks is included in the following section.

B. CIRCUIT COMPONENTS

The circuit shown below is used to test whether a function passes a test for correlation immunity k.

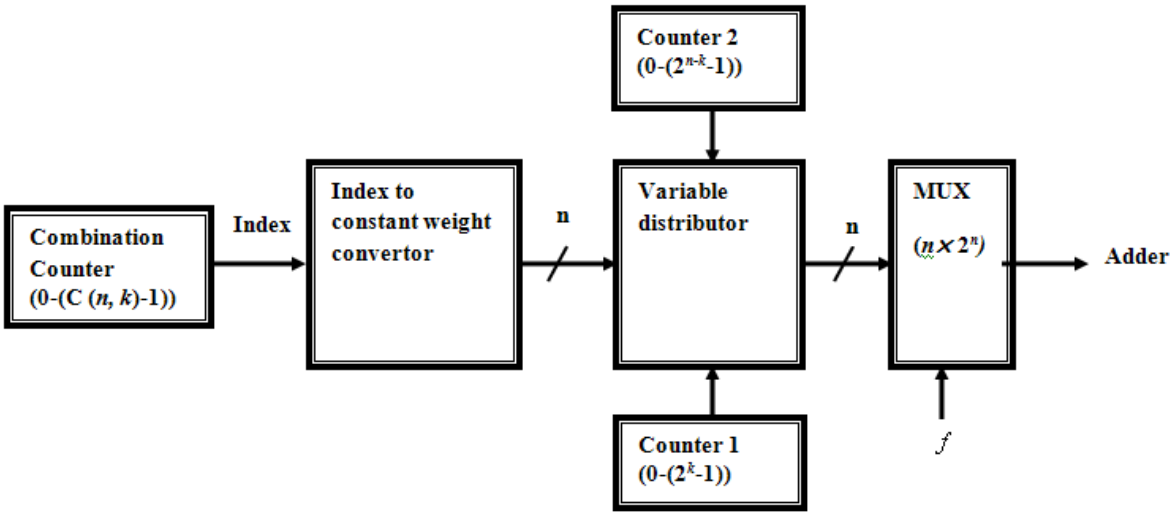

Figure 5. Block diagram of the components used to test a function of *n* variables for correlation immunity *k*.

1. Test for K Blocks

a. Combination Counter

The combination counter enumerates all values for C(n,k) combinations. This block supplies an index to the constant weight convertor. The range of the combination counter is based on the combinatorial number system. In a C(n,k) combinatorial number system, integer N<C(n,k) is represented as N=ckck-1…c1, where $N = C(c_k, k) + C(c_{k-1}, k-1) + ... + C(c_1, 1)$,such that ck>ck-1>…≥0 [10].

Example: The representation of numbers in the C(6,3) combinatorial number system where 0≤N≤19 is shown in Table 4. Each value for N can be transformed into a binary constant weight codeword with k 1s using the definition above. The twenty different values of N correspond to every possible way to distribute k ones over n different positions.

Table 4. The *C(6,3)* combinatorial number system for *0<N<19* from [10].

N	c_1,c_2,c_3 for k=3	Constant Weight Codeword
19	543	111000
18	542	110100
17	541	110010
16	540	110001
15	532	101100
14	531	101010
13	530	101001
12	521	100110
11	520	100101
10	510	100011
9	432	011100
8	431	011010
7	430	011001
6	421	010110
5	420	010101
4	410	010011
3	321	001110
2	320	001101
1	310	001011
0	210	000111

b. Index to Constant Weight Convertor

For each value from the combination counter, the constant weight convertor produces a binary output of size *n* with *k* ones and *(n-k)* zeros. The circuit for an index to constant weight converter for *n*=6 and *k*=3 is shown in Figure 6.

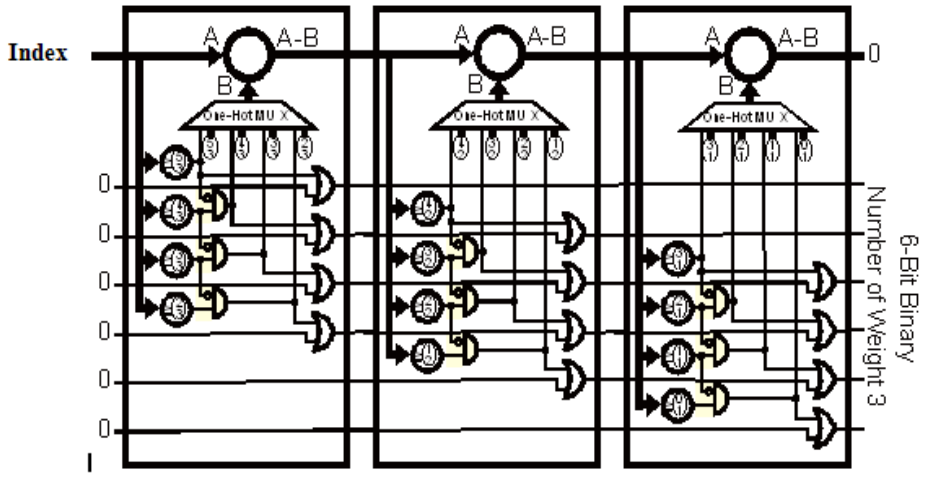

Figure 6. Example of a constant weight codeword generator circuit from [10].

This circuit is made up of a cascade of k lookup tables (LUTs). At the first stage, the index value is compared to n−k+1 values to determine the position of the first 1 in the constant weight codeword. This 1 could be placed in the first through the fourth position in the example shown. In order for a 1 to be placed in the first position, the index must be greater than or equal to C(n−1,k). If the index is not greater than or equal to C(n-1,k), it must be greater than or equal to C(n−2,k) to be placed in the second position. This process continues n-k+1 times. The final comparison will be between the index and the value of C(k−1,k), which is zero. A 1 will be placed in the constant weight codeword at the highest possible position based on the comparisons. The highest value of the comparisons passed will be subtracted from the index value and that value will be used in

the next LUT to determine the placement of the next 1 in the constant weight codeword. At the end of the k LUTs, the index value will be 0 and a constant weight codeword with k 1s will be output.

The 1s in the output represent the selected variables for the test for correlation immunity *k*. In order for a function to have correlation immunity *k*, the function must have the same number of 1s in each subset for all combinations of *k* variables. Each index provides a different way to select *k* variables. Then all possible values of the chosen variables are enumerated in the variable distributor.

c. Variable Distributor

The variable distributor assigns a binary value of n bits that represent an entry in the function's TT. A variable distributor is needed for each value of the two counters. Counter 1 counts from zero to 2k−1. This enumerates all possible binary combinations of the k 1s in the constant codeword. For example, for n=6 and k=3, the counter ranges from 000 to 111. These values are placed in order from most significant to least significant bit in the position corresponding to a 1 in the constant weight codeword. This divides the 2n TT entries into k subsets. Counter 2 counts from zero to (2n-k−1). For each subset created by counter 1, counter 2 enumerates all possible combinations for the other n−k positions. These positions are represented by 0 in the constant weight codeword. For each value of counter 1, counter 2 will cycle through completely. Each time this occurs, one of the 2k subsets is created. The values of each combination from the counter are sent to the multiplexor (MUX).

d. Multiplexor

For each multiplexor (MUX) value, the variable distributor outputs an n bit value that is then applied to an n × 2n MUX that selects the corresponding TT table value of the function under test (FUT).

e. Adder

Once counter 2 enumerates all values, the weight of the subset is calculated and then compared to the number of 1s for subset created by each enumeration of counter 1. If the number of 1s is the same for all subsets, the test is passed for that combination counter value and the combination counter is then incremented and the process is continued. If all tests are passed for that value of *k,* a 1 is set to the priority encoder.

2. Priority Encoder

The priority encoder examines the elements of a vector of length *(n*+1*)* which contains a 1 in the position corresponding to the value of *k* that the function passed all the tests for correlation immunity *k*. The priority encoder selects the position of the highest 1 in the vector and returns the position value as the correlation immunity of the function.

C. PC CIRCUIT

The circuit shown in Figure 7 was implemented in C. It was run on an Intel Xeon processor using a conventional compiler. This C code was also run on the SRC-6's Xilinx Virtex2 Pro FPGA. The SRC-6's compiler can convert C code to Verilog, which creates a circuit on the FPGA. Normally, the C code run on the FPGA is simple code used to support the computation intensive Verilog code that runs on the FPGA. This experiment served to improve understanding of whether computation intensive C code could be effectively compiled into Verilog code.

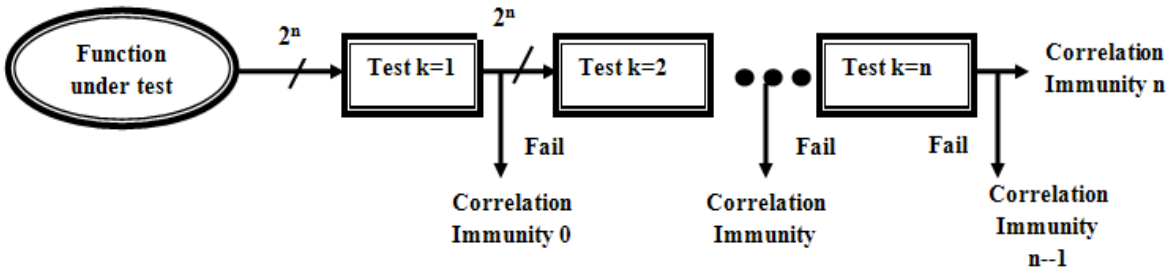

Figure 7. Process for computing the correlation immunity of a function using the PC.

This circuit is similar to the circuit discussed in Chapter III.A except once a function fails a test for correlation immunity it is removed. The tests for *k* blocks are implemented in the same manner and will not be discussed further. This circuit takes advantage of the fact that most Boolean functions have a correlation immunity of 0. With this technique, no unnecessary tests are performed. The disadvantage is the tests are not performed in parallel, which causes a larger delay for functions that pass more tests. Both the SRC-6 and the Intel Xeon processor were less efficient at executing this code than the circuit shown in Figure 1. These results are shown in Chapter IV.B.2 of this thesis. The code is included in Appendix B.

D. SUMMARY

The circuit block diagrams for the circuit used to test a function for correlation immunity written in both Verilog and C programming language were provided. The blocks were discussed in detail to explain the method used to find the correlation immunity.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. RESULTS AND ANALYSIS

A. SRC-6

1. Background

The SRC-6 reconfigurable computer was used for the data collection in this thesis. The SRC-6 allows the user more flexibility to achieve high performance. A microprocessor requires the user to adapt the program to its architecture while the reconfigurable computer allows the user to adapt the computer to their program. By choosing an efficient logic design, the user can achieve optimum performance.

The SRC-6 is composed of two PCs, each with Pentium IV processor and five Multi-Adaptive Processing (MAP) boards. Each MAP contains three high density Xilinx Vertex-2 FPGAs, two of which can programmed and one for control. The SRC-6 has four banks of common memory, 8 GB each.

Multiple files are required to run a program on the SRC-6. Code can be executed to run on one of the two Intel microprocessors associated with SRC-6 or on the MAP. The file structure of a typical project using a user-defined macro is shown in Figure 8. The file *main.c* is written in C code and is run on the Intel processor. The file *main.c* calls a subroutine, which is run on the map. The subroutine, *subr.mc*, is run on the MAP and is also written in C. The *subr.mc* file may make a function call to a Verilog macro, either built-in or created by the user. A makefile is used to control computation and is run on the PC. The makefile will indicate whether a user-defined macro was used in the project. If a user defined macro is included, then three additional files are needed. A black box file, *blk.v*, lists the inputs and outputs to the macro. An information file, *info*, contains the type of macro, length of latency and additional code for debugging purposes. Lastly, the macro file itself is the code used to configure the FPGAs. All three of these files are written in Verilog.

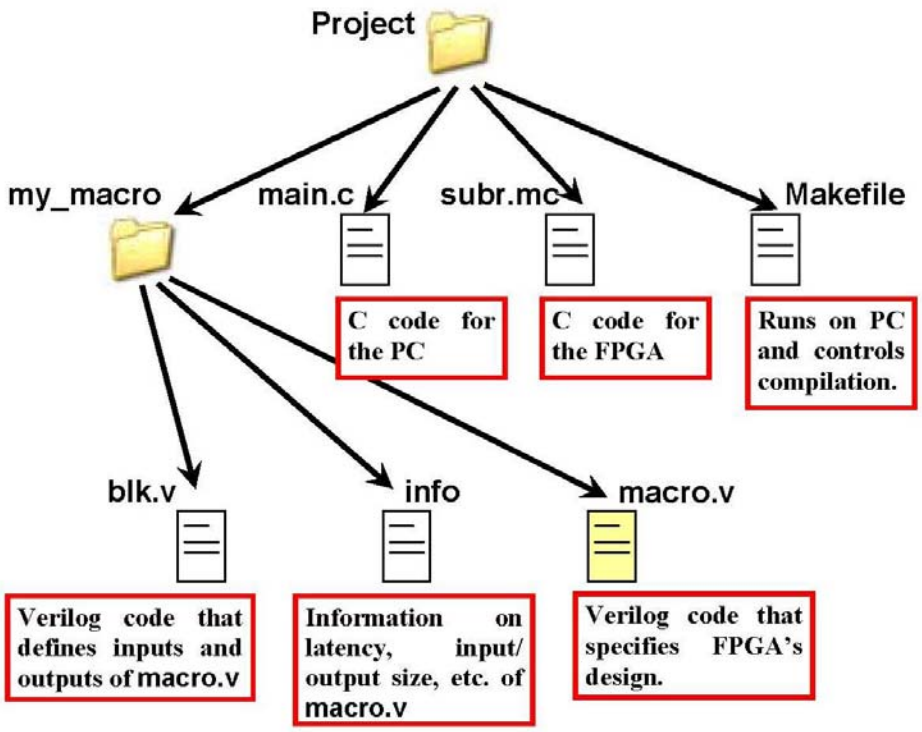

Figure 8. Process for computing the correlation immunity of a function using the PC, from [3].

2. Speed Up

The FPGAs on the SRC-6 have a clock speed of 100 MHz, which is much slower than the 1000-3000 MHz operating speed of a PC. Through sophisticated use of parallelism, even with the much slower clock speed, the SRC-6 is capable of processing information at a much higher speed than the PC.

In order to find the most efficient way to test for the correlation immunity of millions of functions, three different programs were written. One program was written in C code and executed on the Intel Xeon processor at 2.8 GHz. The same program was placed in the *subr.mc* file and run on the MAP. Finally, a macro was written to find the correlation immunity of a given function and run on the MAP. The comparison of computation time for all functions of four variables can be seen in Table 5.

Table 5. Comparison of computation time for all Boolean functions of four variables.

FPGA Compute Time Verilog Macro (@ 100 MHz)	FPGA Compute Time C code (@ 100 MHz)	PC compute Time C code (@ 2.8 GHz)
655.36μsec	1.2387sec	190msec

The SRC-6 program written with a user defined functional macro provided a speed up of almost 300 times that of the PC. This increase is due mostly to the parallel processing ability of the SRC-6, allowing it to test a function for all possible values of correlation immunity in parallel. Pipelining allows the next function to start the testing process as soon as the previous function completes the first testing stage. Once the first function completes the pipeline, results will be output every clock cycle. The PC is not able to achieve this and each function is processed completely before the next function begins. The number of tests for a function of n variables is $2n-1$, so when n increases by one, the number of test doubles. This means as the number of variables increase, the speed up will at least double. The Verilog program performed close to 1900 times faster than the C code executed on the MAP. In this case the compiler translated the C code into Verilog and the FPGAs were configured based on this code. The results shown in Table 5 prove that using a Verilog macro can be much more effective than letting the compiler translate the code itself.

3. Limitations

The main limitation for the SRC-6 is the speed of the FPGA. At 100 MHz, a maximum of 100,000,000 functions can be tested per second. The time it would take to find the correlation immunity of all functions at a rate of one result per clock period is shown in Table 6.

Table 6. Time to calculate the correlation immunity of all Boolean functions.

n	# of Functions	Computation Time for All Boolean Functions @100 MHz
2	16	0.16 μ sec
3	256	2.56 μ sec
4	65536	655.4 μ sec
5	42950×10^9	42.9 sec
6	1.8447×10^{19}	5,849 yrs
7	3.4028×10^{38}	1.1×10^{23} yrs
8	1.1579×10^{77}	3.7×10^{61} yrs
9	1.3408×10^{154}	4.3×10^{138} yrs

It can be seen in Table 6 that at this clock speed the largest exhaustive search that can be performed is for functions of five variables. Based on this, finding smaller subsets that are rich in functions with high correlation immunity is extremely important. A Virtex-5 FPGA runs at 550 MHz which would speed-up the computation time by 5.5. However, even at this speed, an exhaustive search for all functions of six variables is not feasible.

Another limitation is the hardware space on the FPGA. The circuit grows larger as *n* increases and will at some point no longer fit on the FPGA. At this point, the circuit could use an additional FPGA. For this thesis work, only one FPGA was needed.

B. ANALYSIS

1. Balanced Functions

A C code program was written in the *main.c* file to create the set of balanced functions. This program used an index to constant weight convertor to distribute the $2^n/2$ 1s in all possible variations of positions over the *n* variables. The set of balanced functions were then sent in a function call to the *subr.mc* file to find the correlation immunity of each function. The code is included in Appendix A.

a. Correlation Immunity for Balanced Functions for n=4

There are C(16,8) balanced functions for n=4, which is a much smaller space than all Boolean functions for n=4. In order for a function to have a correlation immunity of (n−1), the function must be balanced. The distribution of correlation immunity of balanced functions compared to the distribution of correlation immunity for all functions of four variables is shown in Table 7. The two functions of four variables with correlation immunity 3 are included in this set. The subset also contains all of the functions of four variables with correlation immunity 2. The testing resulted in finding two 3-resilient functions, eight 2-resilient functions, and 212 1-resilient functions.

Table 7. Distribution of correlation immunity for all balanced functions compared to that of all Boolean functions of four variables.

Correlation Immunity	0	1	2	3	4
All Boolean Functions	64,888	636	8	2	2
All Balanced Functions	12,648	212	8	2	0

b. Correlation Immunity for Balanced Functions for n=5

There C(32,16) balanced functions for n=5, which is slightly over 600,000,000 functions. While the SRC-6 can compute a set of this size in six seconds, the code to create balanced functions is more complicated for five variables. By reducing the size of the subset, to include only functions that are balanced over the first sixteen bits and the last sixteen bits, the code was significantly less complex. This reduction did not eliminate any functions with correlation immunity greater than 0. According to the definition of the test for k=1, this smaller subset contains all of the functions with correlation immunity 1 or greater. Therefore, all balanced functions with correlation immunity greater than or equal to 1 can be found by testing this smaller subset. The distribution of correlation immunity for balanced functions compared to the distribution of correlation immunity for all functions of five variables is shown in Table 8. Like the results for $n = 4$, the two functions with correlation immunity n−1 are in the set of balanced functions. Also, the ten functions with correlation immunity n−2 are in the subset of balanced functions. This set also contains a much higher percentage of functions with correlation immunity 1 and 2 than the set of all Boolean functions. The testing resulted in finding two 4-resilient functions, ten 3-resilient functions, 540 2-resilient functions and 807,428 1-resilient functions.

Table 8. Distribution of correlation immunity for all balanced functions compared to that of all Boolean functions of five variables.

Correlation Immunity	0	1	2	3	4	5
All Boolean Functions	4,291,827,234	3,139,004	1044	10	2	2
Balanced Functions subset	164,828,920	807,428	540	10	2	0

2. Rotation Symmetric Functions

Rotation symmetric functions are a very small subset of all functions that has been shown to contain functions that are rich in good cryptographic properties [9]. C code was written in the *main.c* file to create the set of rotation symmetric functions. This was accomplished by creating different sets called orbits, each made up of one cyclic rotation of *n* bits. The value of all TT entries for each orbit may either be set to 1 or 0. The set of rotation symmetric functions is made up of all possible combinations of these orbits being set to 1 or 0. An index to constant weight convertor was used to set the different values of each orbit to 1 or 0 for all possible combinations of the sets. The program first selected zero orbits to create the zero function, then selected all possible ways to choose one orbit to create *n* more functions, then all possible ways to choose two orbits to create $C(n,2)$ more functions, and continued until all orbits were set to 1. The set of rotation symmetric functions were then sent in a function call to the *subr.mc* file to find the correlation immunity of each function. The code is included in Appendix A.

a. Correlation Immunity for Rotation Symmetric Functions for n=4

For n=4, there are 26 rotation symmetric functions [13]. This set contains both of functions with correlation immunity n−1, but none of the functions for n−2. The set does contain a higher percentage of functions with correlation immunity 1 compared to the sample size than the set of all functions. The distribution of correlation immunity of all Boolean functions compared to the distribution of correlation immunity of rotation symmetric functions is shown in Table 9.

Table 9. Distribution of correlation immunity for all rotation symmetric functions compared to that of all Boolean functions of four variables.

Correlation Immunity	0	1	2	3	4
All Boolean Functions	64,888	636	8	2	2
All Rotation Symmetric Functions	48	12	0	2	2

b. Correlation Immunity for Rotation Symmetric Functions for n=5

For n=5, there are 28 rotation symmetric functions which account for 5.9×10^{-8}% of all Boolean functions of five variables. As in the results for n=4, this set contains both of the functions for correlation immunity n−1, but none of the functions for n−2. This set contains a much high percentage of functions for correlation immunity 1 and 2 than the set of all Boolean functions.

Table 10. Distribution of correlation immunity for all rotation symmetric functions compared to that of all Boolean functions of four variables.

Correlation Immunity	0	1	2	3	4	5
All Boolean Functions	4,291,827,234	3,139,004	1044	10	2	2
All Rotation Symmetric Functions	214	34	4	0	2	2

3. Nonlinearity

A macro to find the nonlinearity from [3] was combined with the macro to find the correlation immunity to find both properties of each function. This code is included in Appendix A. Since the pipeline was longer for the nonlinearity circuit than the correlation immunity circuit, a delay module was added to the correlation immunity circuitry. ModelSim™ was used to determine the number of times the delay module would need to be called to have equal pipelines for each circuit. The resulting program output the correlation immunity and nonlinearity of a Boolean function every clock cycle. To verify the results, the C program written to find the correlation immunity was executed with the Verilog macro for nonlinearity. The circuitry was then used to test for the nonlinearity of all functions of a given correlation immunity. This circuit produced the same results.

a. Correlation Immunity and Nonlinearity for All Boolean Functions for n=4

The Distribution of all Boolean functions of four variables by correlation immunity and nonlinearity is shown in Table 11. It can be seen that the highest combination of correlation immunity and nonlinearity is a correlation immunity 1 and nonlinearity of 4. All of the bent functions have correlation immunity 0.

Table 11. Distribution of all Boolean functions of four variables by correlation immunity and nonlinearity.

Nonlinearity/Correlation Immunity	0	1	2	3	4	5
0	8	12	8	2	2	32
1	512	0	0	0	0	512
2	3712	128	0	0	0	3840
3	17920	0	0	0	0	17,920
4	27504	496	0	0	0	28,000
5	14336	0	0	0	0	14,336
6	896	0	0	0	0	896
Total	64,888	636	8	2	2	65,536

b. Correlation Immunity and Nonlinearity for All Boolean Functions for n=5

The distribution of Boolean functions of five variables by correlation immunity is shown in Table 12. For functions with five variables the highest nonlinearity is 12. Based on the results, the highest combination of correlation immunity and nonlinearity that can be achieved for a function of five variables is a nonlinearity of 12 and a correlation immunity of 2. There are 384 functions that meet these criteria.

Table 12. Distribution of Boolean functions of five variables by correlation immunity and nonlinearity.

Nonlinearity/correlation Immunity	0	1	2	3	4	5	6
0	10	20	20	10	2	2	64
1	2,048	0	0	0	0	0	2,048
2	31,232	512	0	0	0	0	31,744
3	317,440	0	0	0	0	0	317,440
4	2,278,400	23,040	0	0	0	0	2,301,440
5	12,888,064	0	0	0	0	0	12,888,064
6	57,873,920	122,368	0	0	0	0	57,996,288
7	215,414,784	0	0	0	0	0	215,414,784
8	645,867,160	1,799,080	640	0	0	0	647,666,880
9	1,362,452,480	0	0	0	0	0	1,362,452,480
10	1,411,209,216	890,880	0	0	0	0	1,412,100,096
11	556,408,832	0	0	0	0	0	556,408,832
12	27,083,648	303,104	384	0	0	0	27,387,136
Total	4,291,827,234	3,139,004	1044	10	2	2	4,294,967,296

4. Correlation Immunity for Functions of Six Variables

The correlation immunity of a random sampling of 2^{16} functions of six variables was found. The SRC-6 has a built in macro that provides a random number generator. This random number generator was used to create two 32 bit numbers that were combined to create the TT value of a function of six variables. The distribution of the correlation immunity for the random sampling of functions of six variables is shown in Table 13. Even with the large sample size, no functions of correlation immunity greater than one were found.

Table 13. Distribution of a random subset of Boolean functions of six variables by correlation immunity.

Correlation Immunity	0	1	2	3	4	5	6
2^{16} Boolean Functions of Six Variables	4,294,850,145	117,151	0	0	0	0	0

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSION AND RECOMMENDATIONS

A. CONCLUSIONS

In this thesis, it was shown that there is a significant benefit to using the SRC-6 reconfigurable computer to test high numbers of Boolean functions for correlation immunity. It was shown that a well-written Verilog macro can be more effective than allowing the compiler to translate C code into Verilog on the SRC-6. Even with the speed-up achieved by using the SRC-6, functions of more than five variables cannot be exhaustively tested using current technology.

To reduce the number of functions that need to be tested to find functions with good correlation immunity, the subsets of balanced functions and rotation symmetric functions were examined and smaller sample sets with functions of high correlation immunity were discovered. The characteristics of functions with higher correlation immunity were examined to find properties to reduce the function space to be tested. The nonlinearity and correlation immunity of all functions of four and five variables were examined to find functions with the highest possible degree of both properties.

B. RECOMMENDATIONS

1. Circular Pipeline

A circular pipeline was developed in another thesis [12] at the Naval Postgraduate School for finding functions with the highest nonlinearity, i.e., bent functions. In order to find a bent function, prior to the creation of the circular pipeline, each function was tested against all affine functions in parallel. The Hamming weight was then calculated for each test, and the minimum was determined. This value is the function's nonlinearity. The functions with the highest weights are called bent functions. Most functions do not have a single bent weight and only need to be tested against one affine function. A speed up of 55 was achieved by using a circular pipeline that removed a function once a test did not result in a bent weight. This work could be expanded on to create a circular pipeline for

correlation immunity. The number of tests passed by all functions for correlation immunity of 1 for n=4 is shown in Figure 9. Most functions do not even pass the first test. A circular pipeline would allow the function to be removed from the pipeline once it fails one test.

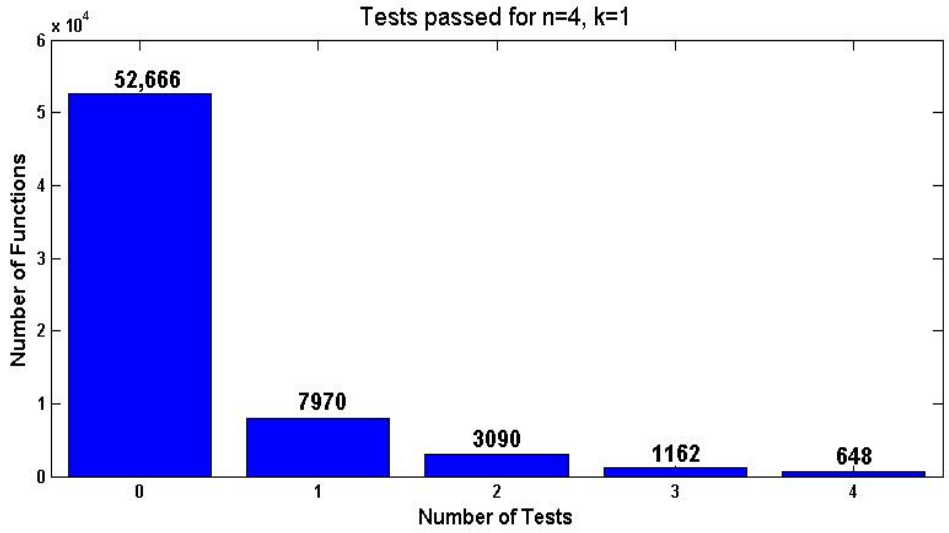

Figure 9. Number of test passed for all functions of four variables for correlation immunity 1.

2. Other Cryptographic Properties

This is the second time the SRC-6 at the Naval Postgraduate School was able to achieve a significant speed-up testing functions for cryptographic properties. A speed-up of 60,000 times was realized using the SRC-6 to test for the nonlinearity of Boolean functions [2]. By combining the code written to find the nonlinearity and the correlation immunity, functions were able to be evaluated for both properties. If more work could be done for other properties, such as algebraic immunity, Boolean functions could be evaluated for all properties in parallel. This could result in finding functions that contained the highest degree of desired characteristics and help to find the best trade-off.

3. Finding Smaller Test Sets

In order to feasibly test all functions for $n=6$ in a reasonable time period, the clock speed would have to be several thousand times faster. Since that may never be technically possible, another solution must to be found. By only looking at balanced functions or rotation symmetric functions, a high percentage of functions with high correlation immunity can be found. By examining the properties of functions with high correlation immunity, we can discover additional subsets that contain high numbers of functions with high correlation immunity.

4. Additional FPGAs

For this thesis, only one FPGA was used. Each MAP in the SRC-6 has three FPGAs, two of which are available for programming. The only obstacle when using two FPGAs is that the bus between the two only allows one 64-bit value to be passed at one time. One way to avoid the problem is to not require communication between the two FPGAs and have both FPGAs implemented to find the correlation immunity of a function. This way, two functions could be tested per clock rather than one.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. SRC-6 CODE

The following code was used to calculate the correlation immunity for the set of all Boolean functions, rotation symmetric functions and balanced functions as well as determining the nonlinearity and correlation immunity of all Boolean functions. Only one makefile is included in the Appendix because it is the same for every program. The *main.c* and subr.mc are the only files listed for the rotation symmetric and balanced programs. These programs used the *blk.v*, info and *corr_imm.v* files for four and five variables listed in Section A and Section B.

A. CORRELATION IMMUNITY FOR N=4

1. main.c

```
/*****************************************************************/
/*  main.c  -a c program designed to run a SRC6 implentaion of    */
/*           corr_imm.v                                           */
/*                                                                */
/*         Authur: Carole Etherington                             */
/*         Last Modified: 04Nov2010                               */
/*                                                                */
/*         Description: This file creates every possible 16 bit   */
/*            binary number then calls a subroutine that returns  */
/*            the corrleation                                     */
/*            immunity of that function.                          */
/*****************************************************************/
#include <map.h>
#include <stdlib.h>

void subr (int64_t*, int64_t*, int64_t*, int);

int main()
{       FILE *res_map,*res_cpu;
        int mapnum=0;
        int n=4;
        int64_t i;
        int64_t time_clk;
        int64_t *x, *ci;
        int count[5];

        x = (int64_t *) malloc (65536* sizeof(int64_t));
        ci = (int64_t *) malloc (65536* sizeof(int64_t));

        for(i=0;i<size;i++)
                {x[i]=i;
```

```
                ci[i]=0;}

        map_allocate(1);
        subr(x,ci,&time_clk,mapnum);
        printf("%lld clocks\n",time_clk);
        for(i=0;i<=n;i++)
            {count[i]=0;}
        for(i=0;i<size;i++)
             {switch(ci[i])
             {case 0: count[0]=count[0]+1;
                 break;
                 case 1: count[1]=count[1]+1;
                 break;
                 case 2: count[2]=count[2]+1;
                 break;
                 case 3: count[3]=count[3]+1;
                 break;
                 case 4: count[4]=count[4]+1;
                 break;
                 default:
                 break;
                }}

                printf("the number of functions with correlation
immunity zero is %lld\n",count[0]);
                printf("the number of functions with correlation
immunity one is %lld\n",count[1]);
                printf("the number of functions with correlation
immunity two is %lld\n",count[2]);
                printf("the number of functions with correlation
immunity three is %lld\n",count[3]);
                printf("the number of functions with correlation
immunity four is %lld\n",count[4]);

        map_free(1);
        exit(0);

        }
```

2. subr.mc

```
/*****************************************************************/
/*  subr.mc  -MAP subroutine to find the correlation immunity of  */
/*            all four variable functions.                       */
/*                                                               */
/*                Author: Carole Etherington                     */
/*          Last modified: November 4, 2010                      */
/*                                                               */
/*          Description: This program calls the macro my_operator */
/*          that finds the correlation immunity of a given function*/
/*          and returns the correlation immunity of the function  */
/*          to the program main.c.                               */
/*                                                               */
/*****************************************************************/
```

```
#include <libmap.h>

void subr (int64_t x[], int64_t ci[], int64_t *time, int mapnum)
{
        OBM_BANK_A(X, int64_t, 65536)
        OBM_BANK_B(CI, int64_t,65536)
        int64_t t0,t1;
        int i;
        int64_t myin;
        int8_t myout;

        DMA_CPU(CM2OBM, X,
        MAP_OBM_stripe(1,"A"),x,1,65536*sizeof(int64_t),0);
        wait_DMA(0);

        read_timer(&t0);

        for(i=0;i<65536;i++)
                {myin=X[i];
                 my_operator(myin, &myout);
                 CI[i]=myout;
                 }

        read_timer(&t1);
        *time=(t1-t0);

DMA_CPU(OBM2CM,CI,MAP_OBM_stripe(1,"B"),ci,1,65536*sizeof(int64_t),0);
   wait_DMA(0);

             }
```

3. makefile

```
# $Id: Makefile.template,v 1.13 2005/04/12 19:18:30 jls Exp $
#
# Copyright 2003 SRC Computers, Inc.  All Rights Reserved.
#
#       Manufactured in the United States of America.
#
# SRC Computers, Inc.
# 4240 N Nevada Avenue
# Colorado Springs, CO 80907
# (v) (719) 262-0213
# (f) (719) 262-0223
#
# No permission has been granted to distribute this software
# without the express permission of SRC Computers, Inc.
#
# This program is distributed WITHOUT ANY WARRANTY OF ANY KIND.
#
# ----------------------------------

# ----------------------------------
# User defines FILES, MAPFILES, and BIN here
```

```
# ----------------------------------
FILES           = main.c

MAPFILES        = subr.mc

BIN             = main

# ----------------------------------
# Multi chip info provided here
# (Leave commented out if not used)
# ----------------------------------
#PRIMARY        = <primary file 1>   <primary file 2>

#SECONDARY      = <secondary file 1> <secondary file 2>

#CHIP2          = <file to compile to user chip 2>

#----------------------------------
# User defined directory of code routines
# that are to be inlined
#----------------------------------
#INLINEDIR      =
# ----------------------------------
# User defined macros info supplied here
#
# (Leave commented out if not used)
# ----------------------------------
MACROS          = my_macro/corr_imm.v
MY_BLKBOX       = my_macro/blk.v
MY_NGO_DIR      = my_macro
MY_INFO         = my_macro/info
# ----------------------------------
# Floating point macros selection
# ----------------------------------
#FPMODE         = SRC_IEEE_V1 # Default SRC version IEEE
#FPMODE         = SRC_IEEE_V2 # Size reduced SRC IEEE with
# special rounding mode
# ----------------------------------
# User supplied MCC and MFTN flags
# ----------------------------------
MCCFLAGS     = -v
MFTNFLAGS    = -v
# ----------------------------------
# User supplied flags for C & Fortran compilers
# ----------------------------------
CC              = gcc   # icc   for Intel cc for Gnu
FC              = ifort # ifort for Intel f77 for Gnu
#LD             = ifort -nofor_main # for mixed C and Fortran, main in
C
#LD             = ifort # for Fortran or C/Fortran mixed, main in
Fortran
LD              = gcc   # for C codes
MY_CFLAGS       =
MY_FFLAGS       =
MY_LDFLAGS      =       # Flags to include libs if needed
```

```
# ----------------------------------
# VCS simulation settings
# (Set as needed, otherwise just leave commented out)
# ----------------------------------
#USEVCS         = yes   # YES or yes to use vcs instead of vcsi
#VCSDUMP        = yes   # YES or yes to generate vcd+ trace dump
# ----------------------------------
# MODELSIM simulation settings
# (Set as needed, otherwise just leave commented out)
# ----------------------------------

#USEMDL         = yes   # YES or yes to use modelsim instead of
vcs/vcsi
#USEMDLGUI      = yes   # YES or yes to use modelsim GUI interface
#MDLDUMP        = yes   # YES or yes to generate vcd trace dump
# ----------------------------------
# No modifications are required below
# ----------------------------------
MAKIN   ?= $(MC_ROOT)/opt/srcci/comp/lib/AppRules.make
include $(MAKIN)
```

4. blk.v

```
/*****************************************************************/
/*  blk.v  -a black-box file that specifies the input/output of   */
/*           corr_imm.v                                           */
/*                                                                */
/*         Authur: Carole Etherington                             */
/*         Last Modified: 04Nov2010                               */
/*****************************************************************/
module corr_imm(TT_ext,CI_ext,CLK);
input CLK;
input[63:0] TT_ext;
output[7:0] CI_ext;
endmodule
```

5. info

```
/*****************************************************************/
/*  info  - This file provides information on the latency, inputs,*/
/*          outputs for the macro, type of macro and output for   */
/*          debugging purposes                                    */
/*         Authur: Carole Etherington                             */
/*         Last Modified: 04Nov2010                               */
/*****************************************************************/
BEGIN_DEF "my_operator"
MACRO= "corr_imm";
STATEFUL =NO;
EXTERNAL =NO;
PIPELINED =YES;
LATENCY =2;

INPUTS=1:
I0=INT 64 BITS (TT_ext[63:0])
```

```
OUTPUTS=1:
O0=INT 8 BITS(CI_ext[7:0]);

IN_SIGNAL: 1 BITS "CLK"="CLOCK";

DEBUG_HEADER =#
        void my_operator__dbg(int64_t TT, int8_t *CI_Ptr);
#;

DEBUG_FUNC=#
        void my_operator__dbg (int64_t TT,int8_t *CI_Ptr)
        {*CI_Ptr=2;}
        #;

      END_DEF
```

6. corr_imm.v

```
module corr_imm (CI_ext, TT_ext,  CLK);
/*****************************************************************/
/*    corr_imm -Verilog code that accepts the truth table, TT, of an */
/*              n-variable function and produces the correlation     */
/*              immunity, C, of that function.                       */
/*     Created:       October 8, 2010                               */
/*     Last Modified: November 4, 2010                              */
/*     Author:        C. Etherington and J. T. Butler               */
/*****************************************************************/
parameter  n = 4;                    // n = number of variables
localparam N = 2**n;
localparam m = clogb2(n);          // m = number of bits to represent n.
wire  [N-1:0]       TT;     // The truth table of the given function.
input   [63:0]       TT_ext;
input                CLK;
wire [m-1:0]       CI;     //   C can be as large as ceil(log_2(n)..
output [7:0]         CI_ext;
wire   [n:0]          k;     //k[i] = 1 iff function has cor. im. at
least i.
genvar i;

generate
  assign k[0]=1'b1; //function will always have at least correlation
immunity zero
   assign TT =TT_ext[N-1:0];

      for (i=1; i<=n; i = i+1)// Enumerate i the index of k to
                              determine highest correlation.
       begin:mult_k
                 cor_im_i #(.n(n),.i(i)) u1 (k[i], TT,CLK);  //k[i]=1
                                          iff TT has cor. im. at least i.
                 end
endgenerate

pri_enc u2 (CI, CLK, k);

assign CI_ext = { {(8-m){1'b0}}, CI };
```

```
//Constant function to find the ceiling of log base two of d
function integer clogb2(input integer d);
      begin
            for(clogb2=0; d>0; clogb2 = clogb2 + 1)
                  d = d >> 1;
      end
endfunction

endmodule

module cor_im_i (k_i, TT,CLK);
/*****************************************************************/
/* corr_imm_i    Verilog code that accepts the truth table, TT, of  */
/*               an n-variable function and produces k_i=1 iff the  */
/*               function has cor. im. at least i, where i is a     */
/*               parameter.  That is, corr_imm_i is a called from a */
/*               generate for loop with index i.  corr_imm_i then   */
/*               enumerates all combinations of i input variables   */
/*               and produces k_i=1 iff for all assignments of      */
/*               values to the i input variables the function has   */
/*               the same number of 1's (and holds for all          */
/*               combinations). This circuit consists of many       */
/*               adders, which add the number of 1's in portions    */
/*               of the truth table of the function. This circuit   */
/*               performs the following sequential code for some    */
/*               combination of i variables indexed by comb         */
/*               (0 <= comb < C(n,i)).                              */
/*     Created:       October 8, 2010                              */
/*     Last Modified: November 22 18, 2010                         */
/*     Author:        C. Etherington and J. T. Butler              */
/*****************************************************************/
parameter n = 4;               // n = the number of variables
parameter i = 2;
localparam N = 2**n;
localparam size=comb_nk(n,i); //finds the number of possible
                              //combinations for choosing
                              //i variables from the number of
                              //variables in the function
input  [N-1:0]        TT;     // The truth table of the given function.
input                 CLK;
output                k_i;    //   C can be as large as ceil(log_2(n)..
reg                   k_i;
reg [n-1:0]      sum [64:0];
integer comb,u,v,cwc;

always @(posedge CLK)

begin
  k_i=1;
  for (comb=0; comb<size; comb=comb+1)  //total number of ways to pick
                                        //i varaibles from n
      begin
       if(k_i==1)
```

```
        begin
        cwc=int2cwc(n,i,comb);
        for (u=0; u<=(2**i-1); u=u+1)  //ennumerates the number of
                                       //subfunctions of size i with
                                       //different values
            begin                        //Scan the 2**i subfunctions.
           sum[u] = 0;              //stores number of ones initially set
                                    //to zero
            for(v=0; v<=2**(n-i)-1; v=v+1) //ennumerates the possible
                                   //values of n-i variables
                 begin
              sum[u] = sum[u] + TT[index(n,i,cwc,u,v)];//totals the
                                    //number of ones for each value of u.
                                   //by finding the fuction value for a
                                  //set u and for each value of v.
                end
           end

        for (u=0; u<2**i-1; u=u+1)
            begin    //Check that all subfunctions  have the same
            if (sum[u] != sum[u+1])//number of 1s.  If not, set k_i=0.
                       k_i = 0
           end

        end
 end
 end
//Constant function
function integer index;//Index to TT.
input integer n;     //Number of variables.
input integer i;     //Prospective cor. im.  (1 <= i <= n)
input integer cwc;    //Index to i-combination. comb[0]=5 comb[1]=4
                        //comb[2]=3 => 111000
input integer u;     //Index to subfunction -            0 <= u < 2^i.
input integer v;       //Index to minterm of subfunction - 0 <= v <
                        //2^(n-i).
integer cwc_temp;
integer u_idx, v_idx, c_idx;
integer u_temp, v_temp;
integer temp;
begin
  u_idx=i-1;
  v_idx=n-i-1;
  index=0;
  u_temp=u;
  v_temp=v;
  cwc_temp=cwc;

   //the following loop finds the truth table index given a set u and v
    //the binary values of u will be place in the corresponding the
binary values of u will be place
    //in the corresponding binary one position of the CWC and the
binary values of v are placed in
    //the binary zero positions of the CWC
    for(c_idx=(n-1);c_idx>=0;c_idx=c_idx-1)
```

```
    begin
      if((cwc_temp-2**c_idx)>=0)   //does the cwc have a one in the
                                    //c_idx position
        begin
          cwc_temp=cwc_temp-2**c_idx;
          if((u_temp-2**u_idx)>=0)//if the u binary value has a one in
                                    //the u_idx position then
              begin               //the index binary value will have a
                                    //one in the cwc_idx position
              index=index+2**c_idx;
              u_temp=u_temp-2**u_idx;
              end
          u_idx=u_idx-1;//one less u binary value to place in index
                        //value
        end
        else //if the cwc has a zero in the c_idx position
          begin
             if((v_temp-2**v_idx)>=0)//if the binary v value has a one
                                    //in the v_idx position
                begin               //then the index value will have a
                                    //one in the cwc_idx position
                index=index+2**c_idx;
                v_temp=v_temp-2**v_idx;
                end
             v_idx=v_idx-1; //one less binary v value to place
           end
      end
end
endfunction

function integer int2cwc(input integer n, input integer i, input
integer comb);
  //functions uses the index value to assign exactly i ones to a n bit
  //string
  integer N,k,comb_temp,temp;
  begin
  N=n-1;// number of positions to place a one from n-1 to zero
  k=i; //number of ones
  comb_temp=comb;
  int2cwc=0;
  while(k>=1//loop continues until all ones have been placed in the
            //string
    begin
      temp=comb_nk(N,k);
      if(comb_temp-temp>=0) //if comb is greater than N choose k
        begin
        int2cwc=int2cwc+2**N;// then place a one in the N position
        comb_temp=comb_temp-temp;
        k=k-1;//decrease number of ones to place
        end
       N=N-1;//decrease the number of positions to place a one.
    end
end
endfunction
```

```
function integer comb_nk(input integer n, input integer k);
integer k_h,k_l,i;
begin:f2
 if(n < k)
 comb_nk = 0;
 else
 begin
 k_h = k;
 k_l = n - k;
 if (k_l > k_h)
 begin
 k_h = n - k;
 k_l = k;
 end
 comb_nk = 1;
 for(i = n; i>k_h; i = i-1)
 comb_nk = comb_nk*i;
 for(i = 1; i<=k_l; i = i+1)
 comb_nk = comb_nk/i;
 end
end
endfunction
endmodule

module pri_enc (CI, CLK,  k);
/*****************************************************************/
/*  pri_enc-Verilog code for a priority encoder. It examines elements*/
/*          of vector k[i], which will be 1 for 1 <= j and 0 for     */
/*          j+1 <= n, in which case the correlation immunity, CI is j*/
/*          and produces the correlation immunity, C, of that        */
/*          function.                                                */
/* Created:       October 8, 2010                                   */
/* Last Modified: October 11, 2010                                  */
/* Author:        C. Etherington and J. T. Butler                   */
/*****************************************************************/
parameter  n = 4;                    // n = the number of variables
localparam m = clogb2(n);
input              CLK;
input  [n:0]       k;//  k[i]=1 means function has cor. im. at least i.
output [m-1:0]     CI;       //  CI can be as large as ceil(log_2(n).
reg    [m-1:0]     CI;
integer            i=1;

always @(posedge CLK)
      begin
      i = 0;      //Set i = 0 and check, for each i from
      while ((i<=n) && (k[i] == 1'b1)) // 0 to n, if k[i]=1.   When
                                       //k[i+1] = 0,
            begin:stage                    // set CI to i.
                  i = i+1
      end
            CI = i-1;
      end

//Constant function
```

```
function integer clogb2(input integer d);
      begin
            for(clogb2=0; d>0; clogb2 = clogb2 + 1)
                  d = d >> 1;
      end
endfunction
endmodule
```

B. CORRELATION IMMUNITY FOR N=5

1. main.c

```
/*****************************************************************/
/*                                                               */
/*  main.c  -a c program designed to run a SRC6 implentaion of   */
/*           corr_imm.v                                          */
/*                                                               */
/*         Authur: Carole Etherington                            */
/*         Last Modified: 15Nov2010                              */
/*                                                               */
/*         Description: This file calls a subroutine that returns */
/*            the corrleation immunity of that function for n=5.  */
/*****************************************************************/
#include <map.h>
#include <stdlib.h>

void subr ( int64_t*, int64_t*, int);

int main()
{       FILE *res_map,*res_cpu;
        int mapnum=0;
        int n=5;
        int64_t i;
        int64_t time_clk;
        int64_t  *ci;
        int count[n];

        ci = (int64_t *) malloc (6* sizeof(int64_t));

        for(i=0;i<6;i++)
                {
                 ci[i]=0;}

        map_allocate(1);
        subr(ci,&time_clk,mapnum);

        printf("%lld clocks\n",time_clk);
```

```
printf("the number of functions with correlation immunity zero is
%lld\n",ci[0]);
printf("the number of functions with correlation immunity one is
%lld\n",ci[1]);
printf("the number of functions with correlation immunity two is
%lld\n",ci[2]);
printf("the number of functions with correlation immunity three is
%lld\n",ci[3]);
printf("the number of functions with correlation immunity three is
%lld\n",ci[4]);
printf("the number of functions with correlation immunity four is
%lld\n",ci[5]);

        map_free(1);
        exit(0);
         }
```

2. subr.mc

```
/*****************************************************************/
/*                                                               */
/*  subr.mc  -MAP subroutine to find the correlation immunity of  */
/*            all five variable functions.                       */
/*                                                               */
/*                Author: Carole Etherington                     */
/*                                                               */
/*          Last modified: November 15, 2010                     */
/*                                                               */
/*                                                               */
/*          Description: This program calls the macro my_operator */
/*          that finds the correlation immunity of a given function*/
/*          and returns the correlation immunity of the function  */
/*          in a histogramto the program main.c.                 */
/*                                                               */
/*****************************************************************/
#include <libmap.h>

void subr ( int64_t ci[], int64_t *time, int mapnum)
{
        OBM_BANK_B(CI, int64_t,6)
        int64_t t0,t1;
        int64_t i,j;
        int k,sel;
        int64_t i0,i1;
        int8_t myout;
        int64_t size;
        int64_t a,b;
        int n=5;
        int64_t H0[5], H1[5],H2[5],H3[5];
        read_timer(&t0);
        k=0;
        for(i=0;i<65536;i++)
                {
                for(j=0;j<65536;j++)
                { i0=i;
```

```
                  i1=j;
                 my_operator(i0,i1,&myout);
                sel=k&3;
                if(sel==0)
                        H0[myout]++;
                if(sel==1)
                        H1[myout]++;
                if(sel==2)
                        H2[myout]++;
                if(sel==3)
                        H3[myout]++;
                k++;
                } }

        for(i=0;i<=n;i++)
         CI[i]=H0[i]+H1[i]+H2[i]+H3[i];


   read_timer(&t1);
   *time=(t1-t0);

   DMA_CPU(OBM2CM,CI,MAP_OBM_stripe(1,"B"),ci,1,6*sizeof(int64_t),0);
   wait_DMA(0);}
```

3. blk.v

```
/*****************************************************************/
/*                                                               */
/*  blk.v  -a black-box file that specifies the input/output of  */
/*           corr_imm.v                                          */
/*                                                               */
/*         Authur: Carole Etherington                            */
/*         Last Modified: 15Nov2010                              */
/*                                                               */
/*****************************************************************/
module corr_imm(TT_ext,TT2_ext,CI_ext,CLK);
input CLK;
input[63:0] TT_ext;
input[63:0] TT2_ext;
output[7:0] CI_ext;

      endmodule
```

4. info

```
/*****************************************************************/
/*                                                               */
/*  info  - This file provides information on the latency, inputs,*/
/*          outputs for the macro, type of macro and output for  */
/*          debugging purposes                                   */
/*         Authur: Carole Etherington                            */
/*         Last Modified: 15Nov2010                              */
/*                                                               */
/*****************************************************************/
BEGIN_DEF "my_operator"
```

```
MACRO= "corr_imm";
STATEFUL =NO;
EXTERNAL =NO;
PIPELINED =YES;
LATENCY =4;

INPUTS=2:
I0=INT 64 BITS (TT_ext[63:0])
I1=INT 64 BITS (TT2_ext[63:0]);

OUTPUTS=1:
O0=INT 8 BITS(CI_ext[7:0]);

IN_SIGNAL: 1 BITS "CLK"="CLOCK";

DEBUG_HEADER =#
        void my_operator__dbg(int64_t TT,int64_t TT2, int8_t *CI_Ptr);
#;
DEBUG_FUNC=#
        void my_operator__dbg (int64_t TT, int64_t TT2,int8_t *CI_Ptr)
        {*CI_Ptr=2;}
        #;

      END_DEF
```

5. corr_imm.v

```
module corr_imm (CI_ext, TT_ext,TT2_ext,  CLK);
/*****************************************************************/
/*    corr_imm -Verilog code that accepts the truth table, TT, of an */
/*              n-variable function and produces the correlation     */
/*              immunity, C, of that function.                       */
/*     Created:       October 8, 2010                               */
/*     Last Modified: November 15, 2010                             */
/*     Author:        C. Etherington and J. T. Butler               */
/*****************************************************************/
parameter  n = 5;                    // n = number of variables
localparam N = 2**n;
localparam m = clogb2(n);        // m = number of bits to represent n.

wire  [N-1:0]       TT;     // The truth table of the given function.
input   [63:0]       TT_ext;
input [63:0]        TT2_ext;

input                CLK;
wire [m-1:0]       CI;     //   C can be as large as ceil(log_2(n)..
output [7:0]         CI_ext;


wire   [n:0]          k//k[i] = 1 iff function has cor. im. at least i.
genvar i;

generate
  assign k[0]=1'b1; //function will always have at least correlation
                  //immunity zero
```

```
   assign TT[31:16] =TT_ext[15:0];
    assign TT[15:0]= TT2_ext[15:0];

        for (i=1; i<=n; i = i+1)// Enumerate i the index of k to
                                //determine highest correlation.
      begin:mult_k
              cor_im_i #(.n(n),.i(i)) u1 (k[i], TT,CLK);  //k[i]=1
                                //iff TT has cor. im. at least i.
              end

endgenerate

pri_enc u2 (CI, CLK, k);

assign CI_ext = { {(8-m){1'b0}}, CI };

//Constant function to find the ceiling of log base two of d
function integer clogb2(input integer d);
        begin
              for(clogb2=0; d>0; clogb2 = clogb2 + 1)
                      d = d >> 1;
        end
endfunction

endmodule

module cor_im_i (k_i, TT,CLK);
/*****************************************************************/
/* corr_imm_i    Verilog code that accepts the truth table, TT, of  */
/*               an n-variable function and produces k_i=1 iff the  */
/*               function has cor. im. at least i, where i is a     */
/*               parameter.  That is, corr_imm_i is a called from a */
/*               generate for loop with index i.  corr_imm_i then   */
/*               enumerates all combinations of i input variables   */
/*               and produces k_i=1 iff for all assignments of      */
/*               values to the i input variables the function has   */
/*               the same number of 1's (and holds for all          */
/*               combinations). This circuit consists of many       */
/*               adders, which add the number of 1's in portions    */
/*               of the truth table of the function. This circuit   */
/*               performs the following sequential code for some   */
/*               combination of i variables indexed by comb        */
/*               (0 <= comb < C(n,i)).                             */
/*     Created:       October 8, 2010                            */
/*     Last Modified: November 15, 2010                          */
/*     Author:        C. Etherington and J. T. Butler             */
/*****************************************************************/
parameter n = 5;              // n = the number of variables
parameter i = 2;
localparam N = 2**n;
localparam size=comb_nk(n,i); //finds the number of possible
                              //combinations for choosing
                              //i variables from the number of
                              //variables in the function
input  [N-1:0]        TT;     // The truth table of the given function.
```

```
input                 CLK;
output                k_i;    //   C can be as large as ceil(log_2(n)..
reg                   k_i;
reg [n-1:0]      sum [64:0];
integer comb,u,v,cwc;

always @(posedge CLK)
begin
  k_i=1;
  for (comb=0; comb<size; comb=comb+1)  //total number of ways to pick
i varaibles from n
      begin
      // if(k_i==1)
       begin
       cwc=int2cwc(n,i,comb);
       for (u=0; u<=(2**i-1); u=u+1)  //ennumerates the number of
                        //subfunctions of size i with different values
           begin        //Scan the 2**i subfunctions.
          sum[u] = 0;    //stores number of ones initially set to zero
           for(v=0; v<=2**(n-i)-1; v=v+1)//ennumerates the possible
                            //values of n-i variables
                begin
                sum[u] = sum[u] + TT[index(n,i,cwc,u,v)];//totals the
                            //number of ones for each value of u.
                           //by finding the fuction value for a set u
                            //and for each value of v.
               end
          end

       for (u=0; u<2**i-1; u=u+1)
           begin    //Check that all subfunctions  have the same
           if (sum[u] != sum[u+1])//  number of 1s.  If not, set k_i=0.
                            k_i = 0;

          end
       end
 end
 end
//Constant function
function integer index;//Index to TT.
input integer n;           //Number of variables.
input integer i;           //Prospective cor. im.  (1 <= i <= n)
input integer cwc;    //Index to i-combination. comb[0]=5 comb[1]=4
                        //comb[2]=3 => 111000
input integer u;           //Index to subfunction  0 <= u < 2^i.
input integer v;  //Index to minterm of subfunction - 0 <= v < 2^(n-i).

integer cwc_temp;
integer u_idx, v_idx, c_idx;
integer u_temp, v_temp;
integer temp;
begin

  u_idx=i-1;
  v_idx=n-i-1;
```

```
  index=0;
  u_temp=u;
  v_temp=v;
 cwc_temp=cwc;

   //the following loop finds the truth table index given a set u and v
    //the binary values of u will be place in the corresponding the
binary values of u will be place
    //in the corresponding binary one position of the CWC and the
binary values of v are placed in
    //the binary zero positions of the CWC
    for(c_idx=(n-1);c_idx>=0;c_idx=c_idx-1)
    begin
      if((cwc_temp-2**c_idx)>=0)//does the cwc have a one in the c_idx
        begin
          cwc_temp=cwc_temp-2**c_idx;
          if((u_temp-2**u_idx)>=0)//if the u binary value has a one in
                                    //the u_idx position then
              begin               //the index binary value will have a
                                    //one in the cwc_idx position
              index=index+2**c_idx;
              u_temp=u_temp-2**u_idx;
              end
          u_idx=u_idx-1;//one less u binary value to place in index
                        //value
        end
        else //if the cwc has a zero in the c_idx position
          begin
             if((v_temp-2**v_idx)>=0)//if the binary v value has a one
                                    //in the v_idx position
                begin               //then the index value will have a
                                    //one in the cwc_idx position
                index=index+2**c_idx;
                v_temp=v_temp-2**v_idx;
                end
             v_idx=v_idx-1; //one less binary v value to place
           end
      end
end
endfunction

function integer int2cwc(input integer n, input integer i, input
integer comb);
//functions uses the index value to assign exactly i ones to a n bit
//string
  integer N,k,comb_temp,temp;
  begin
  N=n-1;// number of positions to place a one from n-1 to zero
  k=i; //number of ones
  comb_temp=comb;
  int2cwc=0;
  while(k>=1) //loop continues until all ones have been placed in the
              //string
    begin
      temp=comb_nk(N,k);
```

```
      if(comb_temp-temp>=0) //if comb is greater than N choose k
        begin
        int2cwc=int2cwc+2**N;// then place a one in the N position
        comb_temp=comb_temp-temp;
        k=k-1;//decrease number of ones to place
        end
       N=N-1;//decrease the number of positions to place a one.
    end
end
endfunction

function integer comb_nk(input integer n, input integer k);
integer k_h,k_l,i;
begin:f2
 if(n < k)
 comb_nk = 0;
 else
 begin
 k_h = k;
 k_l = n - k;
 if (k_l > k_h)
 begin
 k_h = n - k;
 k_l = k;
 end

 comb_nk = 1;
 for(i = n; i>k_h; i = i-1)
 comb_nk = comb_nk*i;
 for(i = 1; i<=k_l; i = i+1)
 comb_nk = comb_nk/i;
 end
end
endfunction

endmodule

module pri_enc (CI, CLK,  k);
/*****************************************************************/
/*  pri_enc-Verilog code for a priority encoder. It examines elements*/
/*          of vector k[i], which will be 1 for 1 <= j and 0 for     */
/*          j+1 <= n, in which case the correlation immunity, CI is j*/
/*          and produces the correlation immunity, C, of that        */
/*          function.                                                */
/* Created:       October 8, 2010                                   */
/* Last Modified: November 15, 2010                                 */
/* Author:        C. Etherington and J. T. Butler                   */
/*****************************************************************/
parameter  n = 5;                        // n = the number of variables
localparam m = clogb2(n);
//
input              CLK;
input  [n:0]       k;                    //  k[i]=1 means function has
cor. im. at least i.
```

```
output [m-1:0]     CI;                  //  CI can be as large as
ceil(log_2(n).
reg    [m-1:0]     CI;
integer            i=1;

always @(posedge CLK)
        begin
        i = 0;    //Set i = 0 and check, for each i from
       while ((i<=n) && (k[i] == 1'b1)) // 0 to n, if k[i]=1.   When
                                        //k[i+1] = 0,
                begin:stage
                        i = i+1;
            end
                CI = i-1;
           end
```

C. CORRELATION IMMUNITY FOR N=6

1. main.c

```
/*****************************************************************/
/*                                                               */
/*  main.c  -a c program designed to run a SRC6 implentaion of   */
/*           corr_imm.v                                          */
/*                                                               */
/*         Authur: Carole Etherington                            */
/*         Last Modified: 22Nov2010                              */
/*                                                               */
/*         Description: This file calls a subroutine that returns */
/*            the corrleation immunity of that function for n=6.  */
/*****************************************************************/
#include <map.h>
#include <stdlib.h>

void subr ( int64_t*, int64_t*, int);

int main()
{       FILE *res_map,*res_cpu;
        int mapnum=0;
        int n=6;
        int64_t i;
        int64_t time_clk;
        int64_t  *ci;
        int count[n];

        ci = (int64_t *) malloc (7* sizeof(int64_t));

        for(i=0;i<7;i++)
                {
                 ci[i]=0;}
```

```
        map_allocate(1);
        subr(ci,&time_clk,mapnum);

        printf("%lld clocks\n",time_clk);
printf("the number of functions with correlation immunity zero is
%lld\n",ci[0]);
printf("the number of functions with correlation immunity one is
%lld\n",ci[1]);
printf("the number of functions with correlation immunity two is
%lld\n",ci[2]);
printf("the number of functions with correlation immunity three is
%lld\n",ci[3]);
printf("the number of functions with correlation immunity three is
%lld\n",ci[4]);
printf("the number of functions with correlation immunity three is
%lld\n",ci[5]);
printf("the number of functions with correlation immunity four is
%lld\n",ci[6]);

        map_free(1);
        exit(0);

        }
```

2. subr.mc

```
/*****************************************************************/
/*                                                               */
/*  subr.mc  -MAP subroutine to find the correlation immunity of  */
/*            all five variable functions.                       */
/*                                                               */
/*                Author: Carole Etherington                     */
/*                                                               */
/*          Last modified: November 22, 2010                     */
/*                                                               */
/*                                                               */
/*          Description: This program calls the macro my_operator */
/*          that finds the correlation immunity of a given function*/
/*          and returns the correlation immunity of the function  */
/*          in a histogramto the program main.c.                 */
/*                                                               */
/*****************************************************************/
#include <libmap.h>
void src_random_32 (int enable, int seed, int reset, int* output);

void subr ( int64_t ci[], int64_t *time, int mapnum)
{
        OBM_BANK_B(CI, int64_t,7)
        int64_t t0,t1;
        int64_t i,j;
        int k,sel;
        int64_t i0,i1;
        int8_t myout;
        int64_t size;
        int64_t a,b;
```

```
        int n=5;
        int64_t H0[5], H1[5],H2[5],H3[5];
        read_timer(&t0);
        k=0;
        for(i=0;i<65536;i++)
                {
                for(j=0;j<65536;j++)
                {i0=src_random_32 (1, seed, i==0, &rndm);
                        i1=src_random_32 (1, seed, j==0, &rndm);
                 my_operator(i0,i1,&myout);
                  sel=k&3;
                if(sel==0)
                        H0[myout]++;
                if(sel==1)
                        H1[myout]++;
                if(sel==2)
                        H2[myout]++;
                if(sel==3)
                        H3[myout]++;
                k++;
                } }

       for(i=0;i<=n;i++)
        CI[i]=H0[i]+H1[i]+H2[i]+H3[i];


   read_timer(&t1);
   *time=(t1-t0);

   DMA_CPU(OBM2CM,CI,MAP_OBM_stripe(1,"B"),ci,1,7*sizeof(int64_t),0);
   wait_DMA(0);
        }
```

3. blk.v

```
/*****************************************************************/
/*                                                               */
/*  blk.v  -a black-box file that specifies the input/output of  */
/*           corr_imm.v                                          */
/*                                                               */
/*         Authur: Carole Etherington                            */
/*         Last Modified: 22Nov2010                              */
/*                                                               */
/*****************************************************************/
module corr_imm(TT_ext,TT2_ext,CI_ext,CLK);
input CLK;
input[63:0] TT_ext;
input[63:0] TT2_ext;
output[7:0] CI_ext;
endmodule
```

4. info

```
/*****************************************************************/
/*                                                               */
```

```
/*  info  - This file provides information on the latency, inputs,*/
/*          outputs for the macro, type of macro and output for   */
/*          debugging purposes                                    */
/*         Authur: Carole Etherington                             */
/*         Last Modified: 22Nov2010                               */
/*                                                                */
/******************************************************************/
BEGIN_DEF "my_operator"
MACRO= "corr_imm";
STATEFUL =NO;
EXTERNAL =NO;
PIPELINED =YES;
LATENCY =4;

INPUTS=2:
I0=INT 64 BITS (TT_ext[63:0])
I1=INT 64 BITS (TT2_ext[63:0]);

OUTPUTS=1:
O0=INT 8 BITS(CI_ext[7:0]);

IN_SIGNAL: 1 BITS "CLK"="CLOCK";

DEBUG_HEADER =#
        void my_operator__dbg(int64_t TT,int64_t TT2, int8_t *CI_Ptr);
#;

DEBUG_FUNC=#
        void my_operator__dbg (int64_t TT, int64_t TT2,int8_t *CI_Ptr)
        {*CI_Ptr=2;}
        #;
END_DEF
```

5. corr_imm.v

```
module corr_imm (CI_ext, TT_ext,TT2_ext,  CLK);
/******************************************************************/
/* corr_imm_i    Verilog code that accepts the truth table, TT, of  */
/*               an n-variable function and produces k_i=1 iff the  */
/*               function has cor. im. at least i, where i is a     */
/*               parameter.  That is, corr_imm_i is a called from a */
/*               generate for loop with index i.  corr_imm_i then   */
/*               enumerates all combinations of i input variables   */
/*               and produces k_i=1 iff for all assignments of      */
/*               values to the i input variables the function has   */
/*               the same number of 1's (and holds for all          */
/*               combinations). This circuit consists of many       */
/*               adders, which add the number of 1's in portions    */
/*               of the truth table of the function. This circuit   */
/*               performs the following sequential code for some   */
/*               combination of i variables indexed by comb         */
/*               (0 <= comb < C(n,i)).                              */
/*     Created:       October 8, 2010                              */
/*     Last Modified: November 22, 2010                            */
/*     Author:        C. Etherington and J. T. Butler              */
```

```
/*****************************************************************/
parameter  n = 6;                    // n = number of variables
localparam N = 2**n;
localparam m = clogb2(n);            // m = number of bits to represent
n.

wire  [N-1:0]       TT;     // The truth table of the given function.
input   [63:0]       TT_ext;
input [63:0]        TT2_ext;

input                CLK;
wire [m-1:0]       CI;     //   C can be as large as ceil(log_2(n)..
output [7:0]         CI_ext;


wire   [n:0]     k;//k[i] = 1 iff function has cor. im. at least i.
genvar i;

generate
  assign k[0]=1'b1;
   assign TT[31:0] =TT_ext[31:0];
    assign TT[63:32]= TT2_ext[31:0];

   for (i=1; i<=n; i = i+1)// Enumerate i the index of k to determine
                        //highest correlation.
       begin:mult_k
                cor_im_i #(.n(n),.i(i)) u1 (k[i], TT,CLK);  //k[i]=1
                             //iff TT has cor. im. at least i.
                end

endgenerate

pri_enc u2 (CI, CLK, k);

assign CI_ext = { {(8-m){1'b0}}, CI };

//Constant function to find the ceiling of log base two of d
function integer clogb2(input integer d);
        begin
                for(clogb2=0; d>0; clogb2 = clogb2 + 1)
                        d = d >> 1;
        end
endfunction
endmodule

module cor_im_i (k_i, TT,CLK);
/*****************************************************************/
/* corr_imm_i    Verilog code that accepts the truth table, TT, of  */
/*               an n-variable function and produces k_i=1 iff the  */
/*               function has cor. im. at least i, where i is a     */
/*               parameter.  That is, corr_imm_i is a called from a */
/*               generate for loop with index i.  corr_imm_i then   */
/*               enumerates all combinations of i input variables   */
/*               and produces k_i=1 iff for all assignments of      */
/*               values to the i input variables the function has   */
```

```
/*               the same number of 1's (and holds for all          */
/*               combinations). This circuit consists of many       */
/*               adders, which add the number of 1's in portions    */
/*               of the truth table of the function. This circuit   */
/*               performs the following sequential code for some    */
/*               combination of i variables indexed by comb         */
/*               (0 <= comb < C(n,i)).                              */
/*     Created:       October 8, 2010                               */
/*     Last Modified: November 22, 2010                             */
/*     Author:        C. Etherington and J. T. Butler               */
/****************************************************************/
parameter n = 6;               // n = the number of variables
parameter i = 2;
localparam N = 2**n;
localparam size=comb_nk(n,i); //finds the number of possible
                              //combinations for choosing
             //i variables from the number of variables in the function
input  [N-1:0]        TT;     // The truth table of the given function.
input                 CLK;
output                k_i;    //   C can be as large as ceil(log_2(n)..
reg                   k_i;
reg [n-1:0]      sum [64:0];
integer comb,u,v,cwc;

always @(posedge CLK)

begin
  k_i=1;
  for (comb=0; comb<size; comb=comb+1)  //total number of ways to pick
                                  //i varaibles from n
      begin
      // if(k_i==1)
       begin
       cwc=int2cwc(n,i,comb);
       for (u=0; u<=(2**i-1); u=u+1)  //ennumerates the number of
                     //subfunctions of size i with different values
           begin                          //Scan the 2**i subfunctions.
          sum[u] = 0;    //stores number of ones initially set to zero
           for(v=0; v<=2**(n-i)-1; v=v+1)//ennumerates the possible
                                  //values of n-i variables
                begin
                sum[u] = sum[u] + TT[index(n,i,cwc,u,v)];//totals the
                  //number of ones for each value of u by finding the
                  //fuction value for a set u and for each value of v.
            end
          end

       for (u=0; u<2**i-1; u=u+1)
           begin      //Check that all subfunctions  have the same
            if (sum[u] != sum[u+1])//  number of 1s.  If not, set
                                  //k_i=0.
             k_i = 0;
        end
        end
 end
```

```
 end
//Constant function
function integer index;//Index to TT.
input integer n;           //Number of variables.
input integer i;           //Prospective cor. im.  (1 <= i <= n)
input integer cwc;    //Index to i-combination.
input integer u;     //Index to subfunction -  0 <= u < 2^i.
input integer v; //Index to minterm of subfunction - 0 <= v < 2^(n-i).
integer cwc_temp;
integer u_idx, v_idx, c_idx;
integer u_temp, v_temp;
integer temp;
begin

  u_idx=i-1;
  v_idx=n-i-1;
  index=0;
  u_temp=u;
  v_temp=v;
 cwc_temp=cwc;

  //the following loop finds the truth table index given a set u and v
  //the binary values of u will be place in the corresponding the
  //binary values of u will be place in the corresponding binary one
  //position of the CWC and the binary values of v are placed in
  //the binary zero positions of the CWC
    for(c_idx=(n-1);c_idx>=0;c_idx=c_idx-1)
    begin
      if((cwc_temp-2**c_idx)>=0)   //does the cwc have a one in the
                                    //c_idx position
        begin
          cwc_temp=cwc_temp-2**c_idx;
          if((u_temp-2**u_idx)>=0)//if the u binary value has a one in
                                    //the u_idx position then
              begin               //the index binary value will have a
                                    //one in the cwc_idx position
              index=index+2**c_idx;
              u_temp=u_temp-2**u_idx;
              end
          u_idx=u_idx-1;//one less u binary value to place in index
        end
        else //if the cwc has a zero in the c_idx position
          begin
             if((v_temp-2**v_idx)>=0)//if the binary v value has a one
                                    //in the v_idx position
                begin       //then the index value will have a one in
                              //the cwc_idx position
                index=index+2**c_idx;
                v_temp=v_temp-2**v_idx;
                end
             v_idx=v_idx-1; //one less binary v value to place
           end
      end
end
endfunction
```

```
function integer int2cwc(input integer n, input integer i, input
integer comb);
  //functions uses the index value to assign exactly i ones to a n bit
string
  integer N,k,comb_temp,temp;
  begin
  N=n-1;// number of positions to place a one from n-1 to zero
  k=i; //number of ones
  comb_temp=comb;
  int2cwc=0;
  while(k>=1)      //loop continues until all ones have been placed in
                  //the string
    begin
      temp=comb_nk(N,k);
      if(comb_temp-temp>=0) //if comb is greater than N choose k
        begin
        int2cwc=int2cwc+2**N;// then place a one in the N position
        comb_temp=comb_temp-temp;
        k=k-1;//decrease number of ones to place
        end
       N=N-1;//decrease the number of positions to place a one.
    end
end
endfunction

function integer comb_nk(input integer n, input integer k);
integer k_h,k_l,i;
begin:f2
 if(n < k)
 comb_nk = 0;
 else
 begin
 k_h = k;
 k_l = n - k;
 if (k_l > k_h)
 begin
 k_h = n - k;
 k_l = k;
 end

 comb_nk = 1;
 for(i = n; i>k_h; i = i-1)
 comb_nk = comb_nk*i;
 for(i = 1; i<=k_l; i = i+1)
 comb_nk = comb_nk/i;
 end
end
endfunction

endmodule

module pri_enc (CI, CLK,  k);
/*****************************************************************/
```

```
/*  pri_enc-Verilog code for a priority encoder. It examines elements*/
/*          of vector k[i], which will be 1 for 1 <= j and 0 for     */
/*          j+1 <= n, in which case the correlation immunity, CI is j*/
/*          and produces the correlation immunity, C, of that        */
/*          function.                                                */
/* Created:       October 8, 2010                                   */
/* Last Modified: November 22, 2010                                 */
/* Author:        C. Etherington and J. T. Butler                   */
/*******************************************************************/
parameter  n = 6;                        // n = the number of variables
localparam m = clogb2(n);
input              CLK;
input  [n:0]       k;//  k[i]=1 means function has cor. im. at least i.
output [m-1:0]     CI;     //  CI can be as large as ceil(log_2(n).
reg    [m-1:0]     CI;
integer            i=1;


always @(posedge CLK)
        begin
        i = 0;          //Set i = 0 and check, for each i from
                while ((i<=n) && (k[i] == 1'b1))
                begin:stage    // set CI to i.
                        i = i+1;
                end
                CI = i-1
        end


//Constant function
function integer clogb2(input integer d);
        begin
                for(clogb2=0; d>0; clogb2 = clogb2 + 1)
                        d = d >> 1;
        end
endfunction
endmodule
```

D. CORRELATION IMMUNITY FOR BALANCED FUNCTIONS, N=5

1. main.c

```
/*******************************************************************/
/*                                                                  */
/*  main.c  -a c program designed to run a SRC6 implentaion of      */
/*           corr_imm.v                                            */
/*                                                                  */
/*         Authur: Carole Etherington                               */
/*         Last Modified: 04Nov2010                                 */
/*                                                                  */
/*         Description: This file creates every possible 16 bit     */
/*            binary number with eight ones and eight zeros and     */
/*            then calls a subroutine that returns the corrleation*/
/*            immunity of that function.                            */
```

```
/*****************************************************************/
#include <map.h>
#include <stdlib.h>

void subr (int64_t*, int64_t*, int64_t*, int);

int main()
{       FILE *res_map,*res_cpu;
        int mapnum=0;
        int n=4;
        int k=8;
        int comb,numer,denom,temp,i,N,comb_temp,a;
        int64_t time_clk;
        int64_t *x, *ci;
        int pow2[16];
        pow2[0]=1;
        pow2[1]=2;
        pow2[2]=4;
        pow2[3]=8;
        pow2[4]=16;
        pow2[5]=32;
        pow2[6]=64;
        pow2[7]=128;
        pow2[8]=256;
        pow2[9]=512;
        pow2[10]=1024;
        pow2[11]=2048;
        pow2[12]=4096;
        pow2[13]=8192;
        pow2[14]=16384;
        pow2[15]=32768;

        x = (int64_t *) malloc (12870* sizeof(int64_t));
        ci = (int64_t *) malloc (6* sizeof(int64_t));

        for(i=0;i<6;i++)
            ci[i]=0;
        //Uses a index to constan weight convertor to place
       //eight ones in a sixteen bit number
        for(i=0;i<12870;i++)
                {
            comb_temp=i;
            N=16-1;//the total number of bits minus one. This number
                  //is increment each time to place either a one or
                  //zero in the 15th to 0 bit position
            x[i]=0;
            k=8;//number of ones the final number will have

            while((N>=0)&&(k>0)) //loop continues until all ones have
                                 been postioned
                {if(N<k)
                temp=0;
                 else
                {numer=1;
                denom=1;
```

```
                for(a=N;a>k;a--)
                        numer=numer*a;
                for(a=(N-k);a>1;a--)
                        denom=denom*a;
                temp=numer/denom;
                } //else
                if((comb_temp-temp)>=0)
                   {x[i]=x[i]+pow2[N];
                    comb_temp=comb_temp-temp;
                    k=k-1;//put one in N bit position and decrease
                            number of ones
                   }//if
                N=N-1;//decrease bit position to be filled
                }//while
        }


        map_allocate(1);
        subr(x,ci,&time_clk,mapnum);

        printf("%lld clocks\n",time_clk);


 printf("the number of functions with correlation immunity zero  is
%lld\n",ci[0]);
 printf("the number of functions with correlation immunity one  is
%lld\n",ci[1]);
 printf("the number of functions with correlation immunity two  is
%lld\n",ci[2]);
 printf("the number of functions with correlation immunity three  is
%lld\n",ci[3]);
 printf("the number of functions with correlation immunity four  is
%lld\n",ci[4]);
 printf("the number of functions with correlation immunity five  is
%lld\n",ci[5]);

        map_free(1);
        exit(0);
        }
```

2. subr.mc

```
/*****************************************************************/
/*                                                               */
/*  subr.mc  -MAP subroutine to find the correlation immunity of  */
/*            all 32 bit balanced functions that have exactly eight*/
/*            ones in the first 16 bits and exactly eight ones in  */
/*            the last sixteen bits.                             */
/*                                                               */
/*                Author: Carole Etherington                     */
/*          Last modified: November 4, 2010                      */
/*          Description: This program calls the macro my_operator  */
/*          that finds the correlation immunity of a given function*/
/*          and returns a histogram of correlation immunity to the */
/*          program main.c.                                      */
```

```
/*****************************************************************/

#include <libmap.h>

void subr (int64_t x[], int64_t ci[], int64_t *time, int mapnum)
{
        OBM_BANK_A(X, int64_t, 12870)
        OBM_BANK_B(CI, int64_t,6)
        int64_t t0,t1;
        int i,j,sel;
        int64_t i0,i1;
        int8_t myout;
        int k=0;
        int64_t H0[6],H1[6],H2[6],H3[6];
        DMA_CPU(CM2OBM, X,
MAP_OBM_stripe(1,"A"),x,1,12870*sizeof(int64_t),0);
        wait_DMA(0);

        read_timer(&t0);
        for(i=0;i<6;i++)
            {H0[i]=0;
             H1[i]=0;
             H2[i]=0;
             H3[i]=0;
            }
      //the nested loop is sent to the macro to be combined to create
    //a 64 bit number. Each X is a 16 bit number that contains eight
ones.
        for(j=0;j<12870;j++)
           for(i=0;i<12870;i++)
                {i0=X[i];
                 i1=X[j];
                 my_operator(i0,i1, &myout);
      //The output alternates between four arrays to prevent slow down
due to
      //read and writes
                          sel=k&3;
                 if(sel==0)
                    H0[myout]++;
                 if(sel==1)
                    H1[myout]++;
                 if(sel==2)
                    H2[myout]++;
                 if(sel==3)
                    H3[myout]++;
                k++;
                }
            //all four values are combined to form the final output
        for(i=0;i<6;i++)
           CI[i]=H0[i]+H1[i]+H2[i]+H3[i];

   read_timer(&t1);
   *time=(t1-t0);

   DMA_CPU(OBM2CM,CI,MAP_OBM_stripe(1,"B"),ci,1,6*sizeof(int64_t),0);
```

```
   wait_DMA(0);
              }
```

E. CORRELATION IMMUNITY FOR ROTATION SYMMETRIC FUNCTIONS, N=4

1. main.c

```
/*****************************************************************/
/*                                                               */
/*  main.c  -a c program designed to run a SRC6 implentaion of   */
/*           corr_imm.v                                          */
/*                                                               */
/*         Authur: Carole Etherington                            */
/*         Last Modified: 04Nov2010                              */
/*                                                               */
/*         Description: This file creates every possible 16 bit  */
/*            rotaionally symmetric number and then calls a      */
/*            subroutine that returns the corrleation            */
/*            immunity of that function.                         */
/*****************************************************************/
#include <map.h>
#include <stdlib.h>

void subr (int64_t*, int64_t*, int64_t*, int);

int main()
{       FILE *res_map,*res_cpu;
        int mapnum=0;
        int n=6;
        int i,k,a,y;
        int numer, denom, j,size, comb, comb_temp,N,cwc,temp;
        int64_t time_clk;
        int64_t *x, *ci;
        int count[5];
        int pow2[16];
        int set[n];
        pow2[0]=1;
        pow2[1]=2;
        pow2[2]=4;
        pow2[3]=8;
        pow2[4]=16;
        pow2[5]=32;
        pow2[6]=64;
        pow2[7]=128;
        pow2[8]=256;
        pow2[9]=512;
        pow2[10]=1024;
        pow2[11]=2048;
        pow2[12]=4096;
```

```
        pow2[13]=8192;
        pow2[14]=16384;
        pow2[15]=32768;


        j=0;
            //The sets of rotionally symmetric numbers
            //If one of the function values in one of the
            //sets is zero then all the values must be zero
            // and if one function value is a one then all
            // function values in the set must be one
        set[0]=pow2[0];
        set[1]=pow2[1]+pow2[8]+pow2[4]+pow2[2];
        set[2]=pow2[3]+pow2[9]+pow2[12]+pow2[6];
        set[3]=pow2[5]+pow2[10];
        set[4]=pow2[7]+pow2[11]+pow2[13]+pow2[14];
        set[5]=pow2[15];



        x = (int64_t *) malloc (64* sizeof(int64_t));
             ci = (int64_t *) malloc (64* sizeof(int64_t));

//This loop creates all the possible combinations of the sets
//given above. It starts with choosing zero sets and producing the
//zero function and then selects one set and then two sets and produces
//all possible combinations by using a index to constant weight
//convertor to place the chosen sets to place a bianry one value in the
//positions of the chosen sets. This continues until the final function
//choses all the sets to have a binary one value.
        for(i=0;i<=6;i++)
                {numer=1;
                 denom=1;
                 for(a=n;a>1;a--)
                    numer=numer*a;
                 for(a=(n-i);a>1;a--)
                    denom=denom*a;
                size=numer/denom;
                for(comb=0;comb<size;comb++)
                   {N=n-1;
                   k=i;
                   comb_temp=comb;
                   cwc=0;
                   while((N>=0)&&(k>0))
                        {if(N<k)
                        temp=0;
                        else
                        {numer=1;
                         denom=1;
                         for(a=N;a>k;a--)
                                numer=numer*a;
                         for(a=(N-k);a>1;a--)
                                denom=denom*a;
                         temp=numer/denom;
                        }
```

```
                    if((comb_temp-temp)>=0)
                            {cwc=cwc+pow2[N];
                            comb_temp=comb_temp-temp;
                            k=k-1;
                            }
                    N=N-1;
                    }
                 x[j]=0;
                 for(y=(n-1);y>=0;y--)
                 {if((cwc-pow2[y])>=0)
                  {x[j]=x[j]+set[y];
                  cwc=cwc-pow2[y];
                  }}


                ci[j]=0;
                j=j+1;
              }}

        map_allocate(1);
        subr(x,ci,&time_clk,mapnum);

        printf("%lld clocks\n",time_clk);

        for(i=0;i<=4;i++)
            {count[i]=0;}

        for(i=0;i<64;i++)
                {switch(ci[i])
                {case 0: count[0]=count[0]+1;
                 break;
                 case 1: count[1]=count[1]+1;
                 break;
                 case 2: count[2]=count[2]+1;
                 break;
                 case 3: count[3]=count[3]+1;
                 break;
                 case 4: count[4]=count[4]+1;
                 break;
                }}

            printf("the number of functions with correlation immunity
zero is %lld\n",count[i]);
            printf("the number of functions with correlation immunity
one is %lld\n",count[i]);
            printf("the number of functions with correlation immunity
two is %lld\n",count[i]);
            printf("the number of functions with correlation immunity
three is %lld\n",count[i]);
            printf("the number of functions with correlation immunity
four is %lld\n",count[i]);

        map_free(1);
        exit(0);
```

```
        }
```

2. subr.mc

```
/*****************************************************************/
/*                                                               */
/*  subr.mc  -MAP subroutine to find the correlation immunity of  */
/*            all four variable functions.                       */
/*                                                               */
/*                Author: Carole Etherington                     */
/*                                                               */
/*          Last modified: November 4, 2010                      */
/*                                                               */
/*                                                               */
/*          Description: This program calls the macro my_operator */
/*          that finds the correlation immunity of a given function*/
/*          and returns the correlation immunity of the function  */
/*          to the program main.c.                               */
/*                                                               */
/*****************************************************************/


#include <libmap.h>

void subr (int64_t x[], int64_t ci[], int64_t *time, int mapnum)
{
        OBM_BANK_A(X, int64_t, 64)
        OBM_BANK_B(CI, int64_t,64)
        int64_t t0,t1;
        int i;
        int64_t myin;
        int8_t myout;

        DMA_CPU(CM2OBM, X,
MAP_OBM_stripe(1,"A"),x,1,64*sizeof(int64_t),0);
        wait_DMA(0);

        read_timer(&t0);

        for(i=0;i<64;i++)
                {myin=X[i];
                 my_operator(myin, &myout);
                 CI[i]=myout;
                 }

   read_timer(&t1);
   *time=(t1-t0);

   DMA_CPU(OBM2CM,CI,MAP_OBM_stripe(1,"B"),ci,1,64*sizeof(int64_t),0);
   wait_DMA(0);
        }
```

F. CORRELATION IMMUNITY FOR ROTATION SYMMETRIC FUNCTIONS, N=5

1. main.c

```
/*****************************************************************/
/*                                                               */
/*  main.c  -a c program designed to run a SRC6 implentaion of   */
/*           corr_imm.v                                          */
/*                                                               */
/*         Authur: Carole Etherington                            */
/*         Last Modified: 04Nov2010                              */
/*                                                               */
/*         Description: This file creates every possible 32 bit  */
/*            rotaionally symmetric number and then calls a      */
/*            subroutine that returns the corrleation            */
/*            immunity of that function.                         */
/*****************************************************************/
#include <map.h>
#include <stdlib.h>

void subr (int64_t*, int64_t*, int64_t*, int);

int main()
{       FILE *res_map,*res_cpu;
        int mapnum=0;
        int n=8;
        int i,k,a,y;
        int numer, denom, j,size, comb, comb_temp,N,cwc,temp;
        int64_t time_clk;
        int64_t *x, *ci;
        int count[6];
        int pow2[16];
        int set1[n];
        int set2[n];
        pow2[0]=1;
        pow2[1]=2;
        pow2[2]=4;
        pow2[3]=8;
        pow2[4]=16;
        pow2[5]=32;
        pow2[6]=64;
        pow2[7]=128;
        pow2[8]=256;
        pow2[9]=512;
        pow2[10]=1024;
        pow2[11]=2048;
        pow2[12]=4096;
        pow2[13]=8192;
        pow2[14]=16384;
        pow2[15]=32768;
```

```
   j=0;

   set1[0]=pow2[0];
   set2[0]=0;
   set1[1]=pow2[1]+pow2[8]+pow2[4]+pow2[2];
   set2[1]=pow2[0];
   set1[2]=pow2[3]+pow2[12]+pow2[6];
   set2[2]=pow2[8]+pow2[1];
   set1[3]=pow2[5]+pow2[10]+pow2[9];
   set2[3]=pow2[4]+pow2[2];
   set1[4]=pow2[7]+pow2[14];
   set2[4]=pow2[12]+pow2[9]+pow2[3];
   set1[5]=pow2[11]+pow2[13];
   set2[5]=pow2[10]+pow2[5]+pow2[6];
   set1[6]=pow2[15];
   set2[6]=pow2[14]+pow2[13]+pow2[11]+pow2[7];
   set1[7]=0;
   set2[7]=pow2[15];
x = (int64_t *) malloc (512* sizeof(int64_t));

ci = (int64_t *) malloc (256* sizeof(int64_t));

   for(i=0;i<=8;i++)
           {numer=1;
            denom=1;
            for(a=n;a>i;a--)
               numer=numer*a;
            for(a=(n-i);a>1;a--)
               denom=denom*a;
           size=numer/denom;
           for(comb=0;comb<size;comb++)
              {N=n-1;
              k=i;
              comb_temp=comb;
              cwc=0;
              while((N>=0)&&(k>0))
                   {if(N<k)
                   temp=0;
                   else
                   {numer=1;
                    denom=1;
                    for(a=N;a>k;a--)
                           numer=numer*a;
                    for(a=(N-k);a>1;a--)
                           denom=denom*a;
                    temp=numer/denom;
                   }
                   if((comb_temp-temp)>=0)
                           {cwc=cwc+pow2[N];
                           comb_temp=comb_temp-temp;
                           k=k-1;
                           }
                   N=N-1;
                   }
               x[j]=0;
```

```
                    x[j+256]=0;
                    for(y=(n-1);y>=0;y--)
                    {if((cwc-pow2[y])>=0)
                     {x[j]=x[j]+set1[y];
                      x[j+256]=x[j+256]+set2[y];
                     cwc=cwc-pow2[y];
                     }}
                   ci[j]=0;
                   j=j+1;
                 }}

        map_allocate(1);
        subr(x,ci,&time_clk,mapnum);

        printf("%lld clocks\n",time_clk);

        for(i=0;i<=5;i++)
            {count[i]=0;}
 for(i=0;i<256;i++)
                {switch(ci[i])
                {case 0: count[0]=count[0]+1;
                 break;
                 case 1: count[1]=count[1]+1;
                 break;
                 case 2: count[2]=count[2]+1;
                 break;
                 case 3: count[3]=count[3]+1;
                 break;
                 case 4: count[4]=count[4]+1;
                 break;
                 case 5: count[5]=count[5]+1;
                 break;
                }}
                for(i=0;i<=5;i++)
                {printf("the number of functions with corrleation
immunity one  is %lld\n",count[i]);}

        map_free(1);
        exit(0);

        }
```

2. subr.mc

```
/*****************************************************************/
/*                                                               */
/*  subr.mc  -MAP subroutine to find the correlation immunity of  */
/*            all four variable functions.                       */
/*                                                               */
/*                Author: Carole Etherington                     */
/*                                                               */
/*          Last modified: November 4, 2010                      */
/*                                                               */
/*                                                               */
/*          Description: This program calls the macro my_operator  */
```

```
/*          that finds the correlation immunity of a given function*/
/*          and returns the correlation immunity of the function   */
/*          to the program main.c.                                 */
/*                                                                 */
/*******************************************************************/
#include <libmap.h>

void subr (int64_t x[], int64_t ci[], int64_t *time, int mapnum)
{
        OBM_BANK_A(X, int64_t, 512)
        OBM_BANK_B(CI, int64_t,256)
        int64_t t0,t1;
        int i;
        int64_t i0,i1;
        int8_t myout;

        DMA_CPU(CM2OBM, X,
MAP_OBM_stripe(1,"A"),x,1,512*sizeof(int64_t),0);
        wait_DMA(0);

        read_timer(&t0);

        for(i=0;i<256;i++)
                {i0=X[i];
                 i1=X[i+256];
                 my_operator(i0,i1, &myout);
                 CI[i]=myout;
                 }

   read_timer(&t1);
   *time=(t1-t0);

   DMA_CPU(OBM2CM,CI,MAP_OBM_stripe(1,"B"),ci,1,256*sizeof(int64_t),0);
   wait_DMA(0);
        }
```

G. CORRELATION IMMUNITY AND NONLINEARITY FOR FUNCTIONS OF N=4

1. main.c

```
/*******************************************************************/
/*                                                                 */
/*  main.c  -a c program designed to run a SRC6 implentaion of     */
/*           corr_imm.v                                            */
/*                                                                 */
/*         Authur: Carole Etherington                              */
/*         Last Modified: 19Nov2010                                */
/*                                                                 */
/*         Description: This file calls a subroutine that returns */
/*            the corrleation immunity and the nonlinearity of     */
/*            of all functions for n=4.                            */
/*******************************************************************/
#include <map.h>
```

```
#include <stdlib.h>

void subr (int64_t*,int64_t*, int64_t*, int64_t*, int);

int main()
{       FILE *res_map,*res_cpu;
        int mapnum=0;
        int n=4;
        int64_t i,j;
        int64_t time_clk;
        int64_t *x, *ci,*out;

        int nonlin[n+1][7];
        int64_t size=65536;
        x = (int64_t *) malloc (size* sizeof(int64_t));
        out =(int64_t *) malloc ( size* sizeof(int64_t));
        ci = (int64_t *) malloc (size* sizeof(int64_t));

        for(i=0;i<size;i++)
                {x[i]=i;
                 ci[i]=0;
                 out[i]=0;}

        map_allocate(1);
        subr(x,ci,out,&time_clk,mapnum);

        printf("%lld clocks\n",time_clk);
        for(i=0;i<=n;i++)
            {for(j=0;j<=6;j++)
            nonlin[i][j]=0;}
        for(i=0;i<size;i++)
                {for(j=0;j<=n;j++)
                {if(ci[i]==j)
                {switch(out[i])
                {
                case 0: nonlin[j][0]=nonlin[j][0]+1;
                break;
                case 1: nonlin[j][1]=nonlin[j][1]+1;
                break;
                case 2: nonlin[j][2]=nonlin[j][2]+1;
                break;
                case 3: nonlin[j][3]=nonlin[j][3]+1;
                break;
                case 4: nonlin[j][4]=nonlin[j][4]+1;
                break;
                case 5: nonlin[j][5]=nonlin[j][5]+1;
                break;
                case 6: nonlin[j][6]=nonlin[j][6]+1;
                break;
                default:
                break;
                }

               }}}
```

```
                for(j=0;j<=6;j++)
                {printf("the number of functions with corrleation
immunity zero  is %lld\n",nonlin[0][j]);
                printf("the number of functions with corrleation
immunity one  is %lld\n",nonlin[1][j]);
                printf("the number of functions with corrleation
immunity two  is %lld\n",nonlin[2][j]);
                printf("the number of functions with corrleation
immunity three  is %lld\n",nonlin[3][j]);
                printf("the number of functions with corrleation
immunity four  is %lld\n",nonlin[4][j]);}

        map_free(1);
        exit(0);

        }
```

2. subr.mc

```
/*****************************************************************/
/*                                                               */
/*  subr.mc  -MAP subroutine to find the correlation immunity of  */
/*            all five variable functions.                       */
/*                                                               */
/*                Author: Carole Etherington                     */
/*                                                               */
/*          Last modified: November 15, 2010                     */
/*                                                               */
/*                                                               */
/*          Description: This program calls the macro my_operator */
/*          that finds the correlation immunity of a given function*/
/*          and returns the correlation immunity of the function  */
/*          in a histogramto the program main.c.                 */
/*                                                               */
/*****************************************************************/
#include <libmap.h>

void subr (int64_t x[], int64_t ci[],int64_t out[], int64_t *time, int
mapnum)
{
        OBM_BANK_A(X, int64_t, 65536)
        OBM_BANK_B(CI, int64_t,65536)
        OBM_BANK_C(OUT,int64_t,65536)
        int64_t t0,t1;
        int i;
        int64_t myin;
        int8_t myout;
        int8_t myciout;

        DMA_CPU(CM2OBM, X,
MAP_OBM_stripe(1,"A"),x,1,65536*sizeof(int64_t),0);
        wait_DMA(0);

        read_timer(&t0);
```

```
        for(i=0;i<65536;i++)
                {
                myin=X[i];
                 my_operator(myin,&myciout, &myout);
                 CI[i]=myciout;
                 OUT[i]=myout;
                 }

   read_timer(&t1);
   *time=(t1-t0);


DMA_CPU(OBM2CM,CI,MAP_OBM_stripe(1,"B"),ci,1,65536*sizeof(int64_t),0);
   wait_DMA(0);

DMA_CPU(OBM2CM,OUT,MAP_OBM_stripe(1,"C"),out,1,65536*sizeof(int64_t),0)
;
   wait_DMA(0);
}
```

3. blk.v

```
/*****************************************************************/
/*                                                               */
/*  blk.v  -a black-box file that specifies the input/output of  */
/*           corr_imm.v                                          */
/*                                                               */
/*         Authur: Carole Etherington                            */
/*         Last Modified: 19Nov2010                              */
/*                                                               */
/*****************************************************************/
module corr_imm(TT_ext,CI_ext,OUT,CLK);
input CLK;
input[63:0] TT_ext;
output[7:0] CI_ext;
output[7:0] OUT;
endmodule
```

4. info

```
/*****************************************************************/
/*                                                               */
/*  info  - This file provides information on the latency, inputs,*/
/*          outputs for the macro, type of macro and output for  */
/*          debugging purposes                                   */
/*         Authur: Carole Etherington                            */
/*         Last Modified: 19Nov2010                              */
/*                                                               */
/*****************************************************************/
BEGIN_DEF "my_operator"
MACRO= "corr_imm";
STATEFUL =NO;
EXTERNAL =NO;
PIPELINED =YES;
LATENCY =5;
```

```
INPUTS=1:
I0=INT 64 BITS (TT_ext[63:0]);

OUTPUTS=2:
O0=INT 8 BITS(CI_ext[7:0])
O1=INT 8 BITS(OUT[7:0]);

IN_SIGNAL: 1 BITS "CLK"="CLOCK";

DEBUG_HEADER =#
        void my_operator__dbg(int64_t TT, int8_t *CI_Ptr, int8_t
*out_Ptr);
#;

DEBUG_FUNC=#
        void my_operator__dbg (int64_t TT, int8_t *CI_Ptr, int8_t
*out_Ptr)
        {*CI_Ptr=2;
         *out_Ptr=2;}
        #;
       END_DEF
```

5. corr_imm.v

```
module corr_imm (CI_ext, OUT, TT_ext,  CLK);
/*****************************************************************/
/* corr_imm_i    Verilog code that accepts the truth table, TT, of  */
/*               an n-variable function and produces k_i=1 iff the  */
/*               function has cor. im. at least i, where i is a     */
/*               parameter.  That is, corr_imm_i is a called from a */
/*               generate for loop with index i.  corr_imm_i then   */
/*               enumerates all combinations of i input variables   */
/*               and produces k_i=1 iff for all assignments of      */
/*               values to the i input variables the function has   */
/*               the same number of 1's (and holds for all          */
/*               combinations). This circuit consists of many       */
/*               adders, which add the number of 1's in portions    */
/*               of the truth table of the function. This circuit   */
/*               performs the following sequential code for some    */
/*               combination of i variables indexed by comb         */
/*               (0 <= comb < C(n,i)). This function also outputs   */
/*               the nonlinearity of a given function               */
/*     Created:       October 8, 2010                             */
/*     Last Modified: November 22, 2010                           */
/*     Author:        C. Etherington and J. T. Butler             */
/*****************************************************************/
parameter  n = 4;                     // n = number of variables
localparam N = 2**n;
localparam m = clogb2(n);         // m = number of bits to represent n.

wire  [N-1:0]       TT;     // The truth table of the given function.
input   [63:0]       TT_ext;
input                CLK;
wire [m-1:0]       CI;     //   C can be as large as ceil(log_2(n)..
```

```
output [7:0]         CI_ext;
output [7:0]        OUT;
wire   [7:0]          l;
wire   [n:0]          k//k[i] = 1 iff function has cor. im. at least i.
wire   [n:0]         temp;
wire   [n:0]         temp2;
wire   [n:0]         temp3;
wire   [n:0]         temp4;
genvar i;

nl_mapper u1(TT,OUT,CLK);

generate
  assign k[0]=1'b1;   assign TT =TT_ext[N-1:0];

  for (i=1; i<=n; i = i+1)// Enumerate i the index of k to determine
                          //highest correlation.
       begin:mult_k
       cor_im_i #(.n(n),.i(i)) u1 (k[i], TT,CLK);  //k[i]=1 iff TT has
                          //cor. im. at least i.
                end
endgenerate

delay u3 (k,temp,CLK);
delay u5 (temp,temp2,CLK);
delay u6 (temp2,temp3,CLK);
delay u7 (temp3,temp4,CLK);
pri_enc u2 (CI, CLK, temp4);

assign CI_ext = { {(8-3){1'b0}}, CI };
//Constant function to find the ceiling of log base two of d
function integer clogb2(input integer d);
        begin
                for(clogb2=0; d>0; clogb2 = clogb2 + 1)
                        d = d >> 1;
        end
endfunction

endmodule

module delay(k,temp,CLK);
parameter  n = 4;
input  [n:0]        k;
output                temp;
reg   [n:0]         temp;
input                 CLK;
always @(posedge CLK)

begin
 temp=k;
 end

endmodule

module cor_im_i (k_i, TT,CLK);
```

```
/*****************************************************************/
/* corr_imm_i    Verilog code that accepts the truth table, TT, of  */
/*               an n-variable function and produces k_i=1 iff the  */
/*               function has cor. im. at least i, where i is a     */
/*               parameter.  That is, corr_imm_i is a called from a */
/*               generate for loop with index i.  corr_imm_i then   */
/*               enumerates all combinations of i input variables   */
/*               and produces k_i=1 iff for all assignments of      */
/*               values to the i input variables the function has   */
/*               the same number of 1's (and holds for all          */
/*               combinations). This circuit consists of many       */
/*               adders, which add the number of 1's in portions    */
/*               of the truth table of the function. This circuit   */
/*               performs the following sequential code for some   */
/*               combination of i variables indexed by comb        */
/*               (0 <= comb < C(n,i)).                             */
/*     Created:       October 8, 2010                             */
/*     Last Modified: November 22, 2010                           */
/*     Author:        C. Etherington and J. T. Butler             */
/*****************************************************************/
parameter n = 4;               // n = the number of variables
parameter i = 2;
localparam N = 2**n;
localparam size=comb_nk(n,i); //finds the number of possible
combinations for choosing
                              //i variables from the number of
variables in the function
input  [N-1:0]        TT;     // The truth table of the given function.
input                 CLK;
output                k_i;    //   C can be as large as ceil(log_2(n)..
reg                   k_i;
reg [n-1:0]      sum [64:0];
integer comb,u,v,cwc;

always @(posedge CLK)

begin
  k_i=1;
  for (comb=0; comb<size; comb=comb+1)  //total number of ways to pick
i varaibles from n
      begin
       //if(k_i==1)
       begin
       cwc=int2cwc(n,i,comb);
       for (u=0; u<=(2**i-1); u=u+1)  //ennumerates the number of
                         //subfunctions of size i with different values
           begin           //Scan the 2**i subfunctions.
          sum[u] = 0;   //stores number of ones initially set to zero
           for(v=0; v<=2**(n-i)-1; v=v+1)//ennumerates the possible
                         //values of n-i variables
                begin
                sum[u] = sum[u] + TT[index(n,i,cwc,u,v)];//totals the
                  //number of ones for each value of u by finding the
                  //fuction value for a set u and for each value of v.
               end
```

```
          end

        for (u=0; u<2**i-1; u=u+1)
            begin      //Check that all subfunctions  have the same
            if (sum[u] != sum[u+1])//  number of 1s.  If not, set k_i=0.
                           k_i = 0;

          end
        end
 end
 end
//Constant function
function integer index;//Index to TT.
input integer n;           //Number of variables.
input integer i;           //Prospective cor. im.  (1 <= i <= n)
input integer cwc;    //Index to i-combination. input integer u;
//Index to subfunction -  0 <= u < 2^i.
input integer v;//Index to minterm of subfunction - 0 <= v < 2^(n-i).
integer cwc_temp;
integer u_idx, v_idx, c_idx;
integer u_temp, v_temp;
integer temp;
begin
  u_idx=i-1;
  v_idx=n-i-1;
  index=0;
  u_temp=u;
  v_temp=v;

   cwc_temp=cwc;
   //the following loop finds the truth table index given a set u and v
  //the binary values of u will be place in the corresponding the
  //binary values of u will be place in the corresponding binary one
  //position of the CWC and the binary values of v are placed in
    //the binary zero positions of the CWC
    for(c_idx=(n-1);c_idx>=0;c_idx=c_idx-1)
    begin
      if((cwc_temp-2**c_idx)>=0)   //does the cwc have a one in the
                                    //c_idx position
        begin
          cwc_temp=cwc_temp-2**c_idx;
          if((u_temp-2**u_idx)>=0)//if the u binary value has a one in
                                    //the u_idx position then
              begin               //the index binary value will have a
                                    //one in the cwc_idx position
              index=index+2**c_idx;
              u_temp=u_temp-2**u_idx;
              end
          u_idx=u_idx-1;//one less u binary value to place in index
                        //value
        end
        else //if the cwc has a zero in the c_idx position
          begin
             if((v_temp-2**v_idx)>=0)//if the binary v value has a one
                                    //in the v_idx position
```

```
                begin               //then the index value will have a
                                    //one in the cwc_idx position
                index=index+2**c_idx;
                v_temp=v_temp-2**v_idx;
                end
              v_idx=v_idx-1; //one less binary v value to place
            end
      end
end
endfunction

function integer int2cwc(input integer n, input integer i, input
integer comb);
  //functions uses the index value to assign exactly i ones to a n bit
string
  integer N,k,comb_temp,temp;
  begin
  N=n-1;// number of positions to place a one from n-1 to zero
  k=i; //number of ones
  comb_temp=comb;
  int2cwc=0;
  while(k>=1//loop continues until all ones have been placed
    begin
      temp=comb_nk(N,k);
      if(comb_temp-temp>=0) //if comb is greater than N choose k
        begin
        int2cwc=int2cwc+2**N;// then place a one in the N position
        comb_temp=comb_temp-temp;
        k=k-1;//decrease number of ones to place
        end
       N=N-1;//decrease the number of positions to place a one.
    end
end
endfunction
function integer comb_nk(input integer n, input integer k);
integer k_h,k_l,i;
begin:f2
 if(n < k)
 comb_nk = 0;
 else
 begin
 k_h = k;
 k_l = n - k;
 if (k_l > k_h)
 begin
 k_h = n - k;
 k_l = k;
 end

 comb_nk = 1;
 for(i = n; i>k_h; i = i-1)
 comb_nk = comb_nk*i;
 for(i = 1; i<=k_l; i = i+1)
 comb_nk = comb_nk/i;
 end
```

```
end
endfunction

endmodule
module pri_enc (CI, CLK,  k);
/*****************************************************************/
/*  pri_enc-Verilog code for a priority encoder. It examines elements*/
/*          of vector k[i], which will be 1 for 1 <= j and 0 for     */
/*          j+1 <= n, in which case the correlation immunity, CI is j*/
/*          and produces the correlation immunity, C, of that        */
/*          function.                                                */
/* Created:       October 8, 2010                                   */
/* Last Modified: November 22, 2010                                 */
/* Author:        C. Etherington and J. T. Butler                   */
/*****************************************************************/
 parameter  n = 4;                        // n = the number of variables
localparam m = clogb2(n);
input              CLK;
input  [n:0]       k;                   //  k[i]=1 means function has
cor. im. at least i.
output [m-1:0]     CI;                  //  CI can be as large as
ceil(log_2(n).
reg    [m-1:0]     CI;
integer            i=1;


always @(posedge CLK)
        begin
        i = 0;   //Set i = 0 and check, for each i from
                while ((i<=n) && (k[i] == 1'b1)) // 0 to n, if k[i]=1.
When k[i+1] = 0,
                begin:stage         // set CI to i.
                        i = i+1;
                end
                CI = i-1;
        end

//Constant function
function integer clogb2(input integer d);
        begin
                for(clogb2=0; d>0; clogb2 = clogb2 + 1)
                        d = d >> 1;
        end
endfunction

endmodule
module nl_mapper(TT,OUT,CLK);
/*****************************************************************/
/* nl_mapper  -  Verilog code to convert the truth table TT of a    */
/*               given function f into a vector, OUT of 2^(n+1)     */
/*               functions - each with 2^n bits that are the        */
/*               distance vectors between f and the 2^(n+1) affine  */
/*               functions.  These are then applied to a ones_count */
/*               circuit to count the number of 1's, which are      */
/*               compared to find the minimum distance from f       */
```

```
/*               to an affine function.                                  */
/*                                                                       */
/*               Created:       November 27, 2008                       */
/*               Last Modified: November 22, 2010                       */
/*               Author:        Jon T. Butler                           */
/* Inputs:        TT   //Truth table of given function, f.               */
/* Outputs:OUT  //Vector of 2^(n+1) distances between f and an afine*/
/*                      function.                                        */
/*************************************************************************/
parameter n = 4;              // n =  number of variables.
localparam N = 2**n;          // N =  number of truth table entries.
localparam NN = 2**(n+1);     // NN = number of affine functions.
input CLK;
wire CLK;
input  [63 : 0]     TT;      // Truth table of function under test.
output [7: 0] OUT;      // Note that TT is unused.  Modify this
  integer i,j,k;       // truth tables of all distance vectors.
reg [N-1:0] temp;
reg [N-1:0] affine;
reg [N-1:0] xored;
reg [NN*(n+1)-1:0] count;

always @(*)                   // truth tables of all affine functions.
 begin
  for (i =0; i<NN; i=i+1)
       begin
       for (j =0; j<N; j=j+1)
           begin
          affine[j]=^(i&((j<<1)+1));
          xored[j]=affine[j]^TT[j];
                   end
                   temp=Count4(xored);
                   for(k=0;k<n+1;k=k+1)
                           begin
                           count[(n+1)*i+k]=temp[k];

                           end
       end

end
min instance_2
(count,OUT,CLK);
function [7:0] Count2;
input [3:0] TT;
   begin: f2
      Count2[0]=TT[3]^TT[2]^TT[1]^TT[0];

Count2[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]);
      Count2[2]=TT[3]&TT[2]&TT[1]&TT[0];
      Count2[3]=1'b0;
      Count2[7:4]=4'b0000;

   end
endfunction
```

```
function [7:0]Count3;
        input [7:0] TT;

        reg [7:0] a,b;
        begin: f3
         a=Count2(TT[3:0]);
         b=Count2(TT[7:4]);
        Count3=a+b;
         end
endfunction
function [7:0]Count4;
        input [15:0] TT;

        reg [7:0] c,d;
        begin: f4
         c=Count3(TT[7:0]);
         d=Count3(TT[15:8]);
           Count4=c+d;
         end
endfunction
function [7:0]Count5;
        input [31:0] TT;

        reg [7:0] e,f;
        begin: f5
         e=Count4(TT[15:0]);
         f=Count4(TT[31:16]);
         Count5=e+f;
         end
endfunction
function [7:0]Count6;
        input [63:0] TT;
        reg [7:0] g,h;
        begin: f6
        g=Count5(TT[31:0]);
        h=Count5(TT[63:32]);
        Count6=g+h;
        end
endfunction
endmodule

module min(IN, OUT, CLK);
/*****************************************************************/
/* min.v  -  A program to compare 2^(n+1) n+1-bit binary values and  */
/*           to deliver the smallest to the output. This can be      */
/*           configured by a Completely pipelined tree               */
/*           In the case of 1.  this runs at 209.6 MHz. for all      */
/*           values of n.  It was tried for n up to 8.   At n=8, it  */
/*           takes more than two minutes to compile.                 */
/*****************************************************************/
parameter  n = 4;       // Number of variables.
localparam nn = n + 1;  // Number of bits in the numbers to be
                        //compared.
localparam N = 2**nn;   // Number of numbers to be compared.  It is the
                                //   number of affine functions.
```

```
output [7:0] OUT;       // OUT is the smallest of the n+1-bit inputs
input [nn*N-1:0]  IN;   // IN is an array of 2^(n+1) (n+1)-bit numbers
        reg  [nn*N-1:0]  curr_IN [nn:0] ;
        input CLK;
      integer i,j;

        always @(posedge CLK)
          begin
            curr_IN[0] <= IN;

            for(j=1; j<=nn; j=j+1)  // Enumerate a level in the
                                    //comparison tree.
              begin
                for(i=0; i<2**(n+1-j); i=i+1) //Enumerate a position in
                                    //the current level.
                  begin:  increment
                        curr_IN[0] <= IN;
     if(curr_IN[j-1][((2*i + 2)*nn-1)-:nn] < curr_IN[j-1][((2*i +
1)*nn-1)-:nn])  curr_IN[j][((i + 1)*nn-1)-:nn] <= curr_IN[j-1][((2*i +
2)*nn-1)-:nn];
                    else  curr_IN[j][((i + 1)*nn-1)-:nn]  <= curr_IN[j-
1][((2*i + 1)*nn-1)-:nn];
                  end

              end


            end
        assign OUT = curr_IN[nn][(nn-1)-:nn];// curr_IN[j][((i + 1)*nn-
1)-:nn] for j=nn and i=0.

endmodule
```

H. CORRELATION IMMUNITY AND NONLINEARITY FOR FUNCTIONS OF N=5

1. main.c

```
/*****************************************************************/
/*                                                               */
/*  main.c  -a c program designed to run a SRC6 implentaion of   */
/*           corr_imm.v                                          */
/*                                                               */
/*         Authur: Carole Etherington                            */
/*         Last Modified: 22Nov2010                              */
/*                                                               */
/*         Description: This file calls a subroutine that returns */
/*            the corrleation immunity and the nonlinearity of   */
/*            of all functions for n=4.                          */
/*****************************************************************/
#include <map.h>
#include <stdlib.h>
```

```
void subr ( int64_t*, int64_t*, int);
int main()
{       FILE *res_map,*res_cpu;
        int mapnum=0;
        int n=5;
        int64_t i,j;
        int64_t time_clk;
        int64_t  *ci;


        ci = (int64_t *) malloc (78* sizeof(int64_t));

        for(i=0;i<78;i++)
                {
                 ci[i]=0;}

        map_allocate(1);
        subr(ci,&time_clk,mapnum);

        printf("%lld clocks\n",time_clk);

        for(i=0;i<78;i++)
        {printf("the number of functions with corrleation immunity one
is %lld\n",ci[i]);}

        map_free(1);
        exit(0);}
```

2. subr.mc

```
/*****************************************************************/
/*                                                               */
/*  subr.mc  -MAP subroutine to find the correlation immunity of  */
/*            all five variable functions.                       */
/*                                                               */
/*                Author: Carole Etherington                     */
/*                                                               */
/*          Last modified: November 22, 2010                     */
/*                                                               */
/*                                                               */
/*          Description: This program calls the macro my_operator */
/*          that finds the correlation immunity of a given function*/
/*          and returns the correlation immunity of the function  */
/*          in a histogramto the program main.c.                 */
/*                                                               */
/*****************************************************************/
#include <libmap.h>

void subr ( int64_t ci[], int64_t *time, int mapnum)
{
        OBM_BANK_B(CI, int64_t,78)
        int64_t t0,t1;
        int64_t i,j;
```

```
        int k,sel;
        int64_t i0,i1;
        int8_t myout;
        int8_t linout;
        int64_t size;
        int64_t a,b;
        int n=5;
        int64_t H0[6][13], H1[6][13],H2[6][13],H3[6][13];
        read_timer(&t0);
        k=0;
       for(i=0;i<6;i++)
          for(j=0;j<13;j++)
           {H0[i][j]=0;
            H1[i][j]=0;
            H2[i][j]=0;
            H3[i][j]=0;}

        for(i=0;i<65536;i++)
                {for(j=0;j<65536;j++)
                { i0=i;
                  i1=j;
                 my_operator(i0,i1,&myout,&linout);


                sel=k&3;
                if(sel==0)
                        H0[myout][linout]++;
                if(sel==1)
                        H1[myout][linout]++;
                if(sel==2)
                        H2[myout][linout]++;
                if(sel==3)
                        H3[myout][linout]++;
                k++;
                } }
       k=0;
       for(i=0;i<=n;i++)
          for(j=0;j<=12;j++)
        {CI[k]=H0[i][j]+H1[i][j]+H2[i][j]+H3[i][j];
        k++;}
 read_timer(&t1);
   *time=(t1-t0);

   DMA_CPU(OBM2CM,CI,MAP_OBM_stripe(1,"B"),ci,1,78*sizeof(int64_t),0);
    wait_DMA(0);}
```

3. blk.v

```
/*****************************************************************/
/*                                                               */
/*  blk.v  -a black-box file that specifies the input/output of  */
/*           corr_imm.v                                          */
/*                                                               */
/*         Authur: Carole Etherington                            */
/*         Last Modified: 19Nov2010                              */
```

```
/*                                                                  */
/********************************************************************/
module corr_imm(TT_ext,TT2_ext,CI_ext,OUT,CLK);
input CLK;
input[63:0] TT_ext;
input[63:0] TT2_ext;
output[7:0] CI_ext;
output[7:0] OUT;
endmodule
```

4. info

```
/********************************************************************/
/*                                                                  */
/*  info  - This file provides information on the latency, inputs,*/
/*          outputs for the macro, type of macro and output for   */
/*          debugging purposes                                    */
/*         Authur: Carole Etherington                             */
/*         Last Modified: 19Nov2010                               */
/*                                                                  */
/********************************************************************/
BEGIN_DEF "my_operator"
MACRO= "corr_imm";
STATEFUL =NO;
EXTERNAL =NO;
PIPELINED =YES;
LATENCY =7;

INPUTS=2:
I0=INT 64 BITS (TT_ext[63:0])
I1=INT 64 BITS (TT2_ext[63:0]);

OUTPUTS=2:
O0=INT 8 BITS(CI_ext[7:0])
O1=INT 8 BITS(OUT[7:0]);
IN_SIGNAL: 1 BITS "CLK"="CLOCK";

DEBUG_HEADER =#
        void my_operator__dbg(int64_t TT,int64_t TT2, int8_t *CI_Ptr,
int8_t *OUT_Ptr);
#;

DEBUG_FUNC=#
        void my_operator__dbg (int64_t TT, int64_t TT2,int8_t
*CI_Ptr,int8_t *OUT_Ptr)
        {*CI_Ptr=2;
         *OUT_Ptr=4;}
        #;
END_DEF
```

5. corr_imm.v

```
module corr_imm (CI_ext, OUT, TT_ext,  CLK);
/********************************************************************/
/* corr_imm_i    Verilog code that accepts the truth table, TT, of  */
```

```
/*               an n-variable function and produces k_i=1 iff the  */
/*               function has cor. im. at least i, where i is a     */
/*               parameter.  That is, corr_imm_i is a called from a */
/*               generate for loop with index i.  corr_imm_i then   */
/*               enumerates all combinations of i input variables   */
/*               and produces k_i=1 iff for all assignments of      */
/*               values to the i input variables the function has   */
/*               the same number of 1's (and holds for all          */
/*               combinations). This circuit consists of many       */
/*               adders, which add the number of 1's in portions    */
/*               of the truth table of the function. This circuit   */
/*               performs the following sequential code for some    */
/*               combination of i variables indexed by comb         */
/*               (0 <= comb < C(n,i)). This function also outputs   */
/*               the nonlinearity of a given function               */
/*     Created:       October 8, 2010                               */
/*     Last Modified: November 22, 2010                             */
/*     Author:        C. Etherington and J. T. Butler               */
/****************************************************************/
parameter  n = 5;                    // n = number of variables
localparam N = 2**n;
localparam m = clogb2(n);            // m = number of bits to represent
n.

wire  [N-1:0]       TT;     // The truth table of the given function.
input   [63:0]       TT_ext;
input [63:0]        TT2_ext;

input                CLK;
wire [m-1:0]       CI;     //   C can be as large as ceil(log_2(n)..
output [7:0]         CI_ext;
output [7:0]        OUT;
wire   [n:0]         k;//k[i] = 1 iff function has cor. im. at least i.
wire   [n:0]         temp;
wire   [n:0]         temp2;
wire   [n:0]         temp3;
wire   [n:0]         temp4;
wire   [n:0]         temp5;
wire    [n:0]        temp6;
genvar i;

nl_mapper u1(TT,OUT,CLK);


generate
  assign k[0]=1'b1
  assign TT[31:16] =TT_ext[15:0];
    assign TT[15:0]= TT2_ext[15:0];


        for (i=1; i<=n; i = i+1)// Enumerate i the index of k to
                               //determine highest correlation.
       begin:mult_k
                cor_im_i #(.n(n),.i(i)) u1 (k[i], TT,CLK);  //k[i]=1
                       //iff TT has cor. im. at least i.
```

```
                end

endgenerate

delay u3 (k,temp,CLK);
delay u5 (temp,temp2,CLK);
delay u6 (temp2,temp3,CLK);
delay u7 (temp3,temp4,CLK);
delay u8 (temp4,temp5,CLK);

pri_enc u2 (CI, CLK, temp5);

assign CI_ext = { {(8-3){1'b0}}, CI };
//Constant function to find the ceiling of log base two of d
function integer clogb2(input integer d);
        begin
                for(clogb2=0; d>0; clogb2 = clogb2 + 1)
                        d = d >> 1;
        end
endfunction

endmodule

module delay(k,temp,CLK);
parameter  n = 5;
input  [n:0]        k;
output                temp;
reg   [n:0]         temp;
input                 CLK;

always @(posedge CLK)
begin
 temp=k;
 end

endmodule

module cor_im_i (k_i, TT,CLK);
/*****************************************************************/
/* corr_imm_i    Verilog code that accepts the truth table, TT, of  */
/*               an n-variable function and produces k_i=1 iff the  */
/*               function has cor. im. at least i, where i is a     */
/*               parameter.  That is, corr_imm_i is a called from a */
/*               generate for loop with index i.  corr_imm_i then   */
/*               enumerates all combinations of i input variables   */
/*               and produces k_i=1 iff for all assignments of      */
/*               values to the i input variables the function has   */
/*               the same number of 1's (and holds for all          */
/*               combinations). This circuit consists of many       */
/*               adders, which add the number of 1's in portions    */
/*               of the truth table of the function. This circuit   */
/*               performs the following sequential code for some   */
/*               combination of i variables indexed by comb        */
/*               (0 <= comb < C(n,i)).                             */
/*     Created:       October 8, 2010                             */
```

```
/*     Last Modified: November 22, 2010                                */
/*     Author:        C. Etherington and J. T. Butler                  */
/*****************************************************************/
parameter n = 5;               // n = the number of variables
parameter i = 2;
localparam N = 2**n;
localparam size=comb_nk(n,i); //finds the number of possible
      //combinations for choosing i variables from the number of
      //variables in the function
input  [N-1:0]        TT;     // The truth table of the given function.
input                 CLK;
output                k_i;    //   C can be as large as ceil(log_2(n)..
reg                   k_i;
reg [n-1:0]      sum [64:0];
integer comb,u,v,cwc;

always @(posedge CLK)

begin
  k_i=1;
  for (comb=0; comb<size; comb=comb+1)  //total number of ways to pick
i varaibles from n
      begin
       //if(k_i==1)
       begin
       cwc=int2cwc(n,i,comb);
       for (u=0; u<=(2**i-1); u=u+1)  //ennumerates the number of
                        //subfunctions of size i with different values
           begin                  //Scan the 2**i subfunctions.
          sum[u] = 0;   //stores number of ones initially set to zero
           for(v=0; v<=2**(n-i)-1; v=v+1)//ennumerates the possible
                        //values of n-i variables
                begin
                sum[u] = sum[u] + TT[index(n,i,cwc,u,v)];//totals the
                        //number of ones for each value of u.
                   //by finding the fuction value for a set u and for
                                //each value of v.
               end
          end

       for (u=0; u<2**i-1; u=u+1)
           begin      //Check that all subfunctions  have the same
           if (sum[u] != sum[u+1])//  number of 1s.  If not, set k_i=0.
                           k_i = 0;

          end
       end
 end
 end

//Constant function
function integer index;//Index to TT.
input integer n;           //Number of variables.
input integer i;           //Prospective cor. im.  (1 <= i <= n)
input integer cwc;    //Index to i-combination.
```

```
input integer u;           //Index to subfunction - 0 <= u < 2^i.
input integer v; //Index to minterm of subfunction - 0 <= v < 2^(n-i).
integer cwc_temp;
integer u_idx, v_idx, c_idx;
integer u_temp, v_temp;
integer temp;
begin
  u_idx=i-1;
  v_idx=n-i-1;
  index=0;
  u_temp=u;
  v_temp=v;

   cwc_temp=cwc;

   //the following loop finds the truth table index given a set u and v
   //the binary values of u will be place in the corresponding the
   //binary values of u will be place in the corresponding binary one
   //position of the CWC and the binary values of v are placed in
    //the binary zero positions of the CWC
    for(c_idx=(n-1);c_idx>=0;c_idx=c_idx-1)
    begin
      if((cwc_temp-2**c_idx)>=0)   //does the cwc have a one in the
                                    //c_idx position
        begin
          cwc_temp=cwc_temp-2**c_idx;
          if((u_temp-2**u_idx)>=0)//if the u binary value has a one in
                                  //the u_idx position then
              Begin //the index binary value will have a one in the
                        //cwc_idx position
              index=index+2**c_idx;
              u_temp=u_temp-2**u_idx;
              end
          u_idx=u_idx-1;//one less u binary value to place in index
                        //value
        end
        else //if the cwc has a zero in the c_idx position
          begin
             if((v_temp-2**v_idx)>=0)//if the binary v value has a one
                                    //in the v_idx position
                begin               //then the index value will have a
                                    //one in the cwc_idx position
                index=index+2**c_idx;
                v_temp=v_temp-2**v_idx;
                end
             v_idx=v_idx-1; //one less binary v value to place
           end
      end
end
endfunction

function integer int2cwc(input integer n, input integer i, input
integer comb);
  //functions uses the index value to assign exactly i ones to a n bit
  //string
```

```
  integer N,k,comb_temp,temp;
  begin
  N=n-1;// number of positions to place a one from n-1 to zero
  k=i; //number of ones
  comb_temp=comb;
  int2cwc=0;
  while(k>=1)//loop continues until all ones have been placed    begin
      temp=comb_nk(N,k);
      if(comb_temp-temp>=0) //if comb is greater than N choose k
        begin
        int2cwc=int2cwc+2**N;// then place a one in the N position
        comb_temp=comb_temp-temp;
        k=k-1;//decrease number of ones to place
        end
       N=N-1;//decrease the number of positions to place a one.
    end
end
endfunction

function integer comb_nk(input integer n, input integer k);
integer k_h,k_l,i;
begin:f2
 if(n < k)
 comb_nk = 0;
 else
 begin
 k_h = k;
 k_l = n - k;
 if (k_l > k_h)
 begin
 k_h = n - k;
 k_l = k;
 end

 comb_nk = 1;
 for(i = n; i>k_h; i = i-1)
 comb_nk = comb_nk*i;
 for(i = 1; i<=k_l; i = i+1)
 comb_nk = comb_nk/i;
 end
end
endfunction

endmodule
module pri_enc (CI, CLK,  k);
/*****************************************************************/
/*  pri_enc-Verilog code for a priority encoder. It examines elements*/
/*          of vector k[i], which will be 1 for 1 <= j and 0 for     */
/*          j+1 <= n, in which case the correlation immunity, CI is j*/
/*          and produces the correlation immunity, C, of that        */
/*          function.                                                */
/* Created:       October 8, 2010                                   */
/* Last Modified: November 22, 2010                                 */
/* Author:        C. Etherington and J. T. Butler                   */
/*****************************************************************/
```

```
 parameter  n = 5;                        // n = the number of variables
localparam m = clogb2(n);
input              CLK;
input  [n:0]       k;                    //  k[i]=1 means function has
cor. im. at least i.
output [m-1:0]     CI;                   //  CI can be as large as
ceil(log_2(n).
reg    [m-1:0]     CI;
integer            i=1;
always @(posedge CLK)
        begin
        i = 0;   //Set i = 0 and check, for each i from
                while ((i<=n) && (k[i] == 1'b1))
begin:stage         // set CI to i.
                        i = i+1;
                end
                CI = i-1;
        end

//Constant function
function integer clogb2(input integer d);
        begin
                for(clogb2=0; d>0; clogb2 = clogb2 + 1)
                        d = d >> 1;
        end
endfunction

endmodule
module nl_mapper(TT,OUT,CLK);
/*****************************************************************/
/* nl_mapper  -  Verilog code to convert the truth table TT of a    */
/*               given function f into a vector, OUT of 2^(n+1)     */
/*               functions - each with 2^n bits that are the        */
/*               distance vectors between f and the 2^(n+1) affine  */
/*               functions.  These are then applied to a ones_count */
/*               circuit to count the number of 1's, which are      */
/*               compared to find the minimum distance from f       */
/*               to an affine function.                             */
/*                                                                  */
/*               Created:       November 27, 2008                   */
/*               Last Modified: November 22, 2010                   */
/*               Author:        Jon T. Butler                       */
/* Inputs:        TT   //Truth table of given function, f.          */
/* Outputs:OUT  //Vector of 2^(n+1) distances between f and an afine*/
/*                       function.                                  */
/*****************************************************************/
parameter n = 5;              // n =  number of variables.
localparam N = 2**n;          // N =  number of truth table entries.
localparam NN = 2**(n+1);     // NN = number of affine functions.
input CLK;
wire CLK;
input  [63 : 0]     TT;      // Truth table of function under test.
output [7: 0] OUT;      // Note that TT is unused.  Modify this
  integer i,j,k;       // truth tables of all distance vectors.
reg [N-1:0] temp;
```

```
reg [N-1:0] affine;
reg [N-1:0] xored;
reg [NN*(n+1)-1:0] count;

always @(*)                    // truth tables of all affine functions.
 begin
  for (i =0; i<NN; i=i+1)
       begin
       for (j =0; j<N; j=j+1)
           begin
          affine[j]=^(i&((j<<1)+1));
          xored[j]=affine[j]^TT[j];
                   end
                   temp=Count4(xored);
                   for(k=0;k<n+1;k=k+1)
                           begin
                           count[(n+1)*i+k]=temp[k];

                           end
       end

end
min instance_2
(count,OUT,CLK);
function [7:0] Count2;
input [3:0] TT;
   begin: f2
      Count2[0]=TT[3]^TT[2]^TT[1]^TT[0];

Count2[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]);
      Count2[2]=TT[3]&TT[2]&TT[1]&TT[0];
      Count2[3]=1'b0;
      Count2[7:4]=4'b0000;

   end
endfunction
function [7:0]Count3;
        input [7:0] TT;

        reg [7:0] a,b;
        begin: f3
         a=Count2(TT[3:0]);
         b=Count2(TT[7:4]);
        Count3=a+b;
         end
endfunction
function [7:0]Count4;
        input [15:0] TT;

        reg [7:0] c,d;
        begin: f4
         c=Count3(TT[7:0]);
         d=Count3(TT[15:8]);
           Count4=c+d;
```

```
         end
endfunction
function [7:0]Count5;
        input [31:0] TT;

        reg [7:0] e,f;
        begin: f5
         e=Count4(TT[15:0]);
         f=Count4(TT[31:16]);
         Count5=e+f;
         end
endfunction
function [7:0]Count6;
        input [63:0] TT;
        reg [7:0] g,h;
        begin: f6
        g=Count5(TT[31:0]);
        h=Count5(TT[63:32]);
        Count6=g+h;
        end
endfunction
endmodule

module min(IN, OUT, CLK);
/*****************************************************************/
/* min.v  -  A program to compare 2^(n+1) n+1-bit binary values and  */
/*           to deliver the smallest to the output. This can be      */
/*           configured by a Completely pipelined tree               */
/*           In the case of 1.  this runs at 209.6 MHz. for all      */
/*           values of n.  It was tried for n up to 8.   At n=8, it  */
/*           takes more than two minutes to compile.                 */
/*****************************************************************/
parameter  n = 5;       // Number of variables.
localparam nn = n + 1;  // Number of bits in the numbers to be
                        //compared.
localparam N = 2**nn;   // Number of numbers to be compared.  It is the
                               //   number of affine functions.
output [7:0] OUT;       // OUT is the smallest of the n+1-bit inputs
input [nn*N-1:0]  IN;   // IN is an array of 2^(n+1) (n+1)-bit numbers
        reg  [nn*N-1:0]  curr_IN [nn:0] ;
        input CLK;
      integer i,j;

        always @(posedge CLK)
          begin
            curr_IN[0] <= IN;

            for(j=1; j<=nn; j=j+1)  // Enumerate a level in the
                                    //comparison tree.
              begin
                for(i=0; i<2**(n+1-j); i=i+1) //Enumerate a position in
                                //the current level.
                 begin:  increment
                       curr_IN[0] <= IN;
```

```
     if(curr_IN[j-1][((2*i + 2)*nn-1)-:nn] < curr_IN[j-1][((2*i +
1)*nn-1)-:nn])  curr_IN[j][((i + 1)*nn-1)-:nn] <= curr_IN[j-1][((2*i +
2)*nn-1)-:nn];
                    else  curr_IN[j][((i + 1)*nn-1)-:nn]  <= curr_IN[j-
1][((2*i + 1)*nn-1)-:nn];
                  end

             end


           end
        assign OUT = curr_IN[nn][(nn-1)-:nn];// curr_IN[j][((i + 1)*nn-
1)-:nn] for j=nn and i=0.

endmodule
```

APPENDIX B. PC CODE

The following is the program used to find the correlation immunity on the PC.

A. CORRELATION IMMUNITY FOR N=4

1. Main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main () {
                int pow2[16];
                int x;
                int
i,comb,u,v,a,u_temp,v_temp,v_idx,u_idx,N,k,cwc_temp;
                int comb_temp,cwc,numer,denom,temp,index,c_idx,size;
                clock_t start,stop;
                double duration;
                int sum[17];
                int ci;
                int k_i[5];
                int count[5];
                int n=4;
                pow2[0]=1;
                pow2[1]=2;
                pow2[2]=4;
                pow2[3]=8;
                pow2[4]=16;
                pow2[5]=32;
                pow2[6]=64;
                pow2[7]=128;
                pow2[8]=256;
                pow2[9]=512;
                pow2[10]=1024;
                pow2[11]=2048;
                pow2[12]=4096;
                pow2[13]=8192;
                pow2[14]=16384;
                pow2[15]=32768;


for(x=0;x<=65536;x++)
count[x]=0;

start=clock();
for(x=0;x<10;x++)
    {k_i[0]=1;
    for(i=1;i<=n;i++)
```

```
     {if(k_i[i-1]==1)
       {numer=1;
        denom=1;
         for(a=n;a>i;a--)
            numer=a*numer;
             for(a=(n-i);a>1;a--)
                         denom=denom*a;
                         size=numer/denom;
                         k_i[i]=1;
                   for(comb=0;comb<size;comb++)
                      {if(k_i[i]==1)
                       {N=n-1;
                        k=i;
                        comb_temp=comb;
                        cwc=0;
                        while((N>=0) && (k>0))
                            {if(N<k)
                              temp=0;
                              else
                              {numer=1;
                               denom=1;
                               for(a=N;a>k;a--)
                                   numer=a*numer;
                               for(a=(N-k);a>1;a--)
                                   denom=denom*a;
                                   temp=numer/denom;
                                    }//else
                               if((comb_temp-temp)>=0)
                                    {cwc=cwc+pow2[N];
                                      comb_temp=comb_temp-temp;
                                      k=k-1;
                                       }//if statement
                                         N=N-1;}
                          for(u=0;u<=(pow2[i]-1);u++)
                                {sum[u]=0;
                                 for(v=0;v<=(pow2[n-i]-1);v++)
                                 {u_idx=i-1;
                                  v_idx=n-i-1;
                                  index=0;
                                  u_temp=u;
                                  v_temp=v;
                                  cwc_temp=cwc;

                                         for(c_idx=(n-1);c_idx>=0;c_idx-
-)
                                         {if((cwc_temp-pow2[c_idx])>=0)

                                                 {cwc_temp=cwc_temp-
pow2[c_idx];
                                                 if((u_temp-
pow2[u_idx])>=0)
{index=index+pow2[c_idx];
```

```
                                                        u_temp=u_temp-
pow2[u_idx];
                                                      }//if statement
                                                   u_idx=u_idx-1;
                                               }//if cwc statement
                                           else
                                                  {if((v_temp-
pow2[v_idx])>=0)

{index=index+pow2[c_idx];
                                                                v_temp=v_temp-
pow2[v_idx];
                                                   }//if statement
                                               v_idx=v_idx-1;
                                               }//else statement
                                           }//for c_idx loop

if((x&pow2[index])==pow2[index])
                                           sum[u]=sum[u]+1;

                                         }//v loop
                                   }//uloop



                                   for(u=0;u<pow2[i]-1;u++)
                                           {if(sum[u]!= sum[u+1])
                                           {k_i[i]=0;
                                           break;
                                           }}
}}}}// if statement (k=1) and for comb and i loops
ci=0;
for(i=1;i<=n;i++)
        {if(k_i[i]==1)
                ci=ci+1;
        else
                break;
        }//for loop


    {switch(ci)
    {case 0: count[0]=count[0]+1;
    break;
     case 1: count[1]=count[1]+1;
     break;
     case 2: count[2]=count[2]+1;
     break;
     case 3: count[3]=count[3]+1;
     break;
     case 4: count[4]=count[4]+1;
     break;
     case 5: count[5]=count[5]+1;
     break;
     default:
     break;
```

```
    }
   }
} //x loop
stop=clock();
duration =(double)(stop-start)/CLOCKS_PER_SEC;
printf("\nThe number of seconds was %.40f\n",duration);


        printf("Histogram of maximum k for which functions have
correlation immunity k \n");

               for(x=0;x<=n;x++)
                printf("\n n=%2d\n", count[x]);

 }//int main (int argc, char *argv[]) {
```

LIST OF REFERENCES

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

[2] J. L. Shafer, S.W. Schneider, J.T. Butler, and P. Stanica, "Enumeration of bent Boolean functions by reconfigurable computer," in *18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines,* pp. 265–272, 2010.

[3] "Introduction to EC3820 and its Laboratory," class notes for EC3820, Department of Electrical and Computer Engineering, Naval Postgraduate School, Fall 2010.

[4] SRC Computers, Inc., "SRC Carte™ C Programming Environment v3.2 Guide," SRC-007-20, Colorado Springs, Colorado, November 2009.

[5] E. Filiol and C. Fontaine, "Highly nonlinear balanced Boolean functions with a good correlation-immunity," *Advances in Cryptology,* vol. 1403, pp. 475–488, 1998.

[6] Y. Yang, "Correlation-immunity of Boolean functions," *Electronic Letters*, vol. 23, pp. 1335–1336, 1987.

[7] C. Charlet, "On bent and highly nonlinear balanced/resilient functions and their algebraic immunities," *Applied Algebra, Algebraic Algorithms and Their Algebraic Immunities,* vol. 3857, pp 1–28, 2006.

[8] T. Siegenthaler, "Correlation-immunity of nonlinear combining functions for cryptographic applications," *IEEE Transactions on Information Theory,* vol. IT-30, pp 776–780, 1984.

[9] D. Dalai, S. Maitra, and S. Sarkar, "Results on rotation symmetric bent functions," *Discrete Mathematics,* vol. 309, pp 2398–2409, 2009.

[10] J. T. Butler and T. Sasao, "Index to constant weight code converter," October 2010 Preprint.

[11] J. T. Butler, "Computing correlation immunity by reconfigurable computer," June 2010 Preprint.

[12] C. Johnson, "The circular pipeline: Achieving higher throughput in the search for bent functions," M.S. thesis, Naval Postgraduate School, Monterey, CA, 2010.

[13] P. Stanica and S. Maitra, "Rotation symmetric functions-count and cryptographic properties," Discrete Applied Mathematics, 156.10, pp 1567–1580, 2008.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Dr. Clark Robertson
   Naval Postgraduate School
   Monterey, California

4. Dr. John G. Harkins
   National Security Agency
   Fort Meade, Maryland

5. Dr. David R. Podany
   National Security Agency
   Fort Meade, Maryland

6. Mr. David Caliga
   SRC Computers
   Colorado Springs, Colorado

7. Mr. Jon Huppenthal
   SRC Computers
   Colorado Springs, Colorado

8. Dr. Jeff Hammes
   SRC Computers
   Colorado Springs, Colorado

9. Dr. Jon T. Butler
   Naval Postgraduate School
   Monterey, California

10. Dr. Pantelimon Stanica
    Naval Postgraduate School
    Monterey, California

11. Mr. Christopher Johnson
Naval Postgraduate School
Monterey, California

12. Dr. Robert L. Herklotz
Program Manager, Information Operations and Security
Air Force Office of Scientific Research (AFROSR/RSL)
Arlington, Virginia