The Classification Of Boolean Functions Using The Rademacher-Walsh Transform

by

Neil Arnold Anderson
B.Sc, University Of Lethbridge, 2004

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF SCIENCE

in the Department Of Computer Science

Supervisory Committee

The Classification Of Boolean Functions Using The Rademacher-Walsh Transform

by

Neil Anderson
Bachelor of Science, University of Lethbridge, 2004

Abstract

When considering Boolean switching functions with n input variables, there are 2^{2^n} possible functions that can be realized by enumerating all possible combinations of input values and arrangements of output values. As is expected with double exponential growth, the number of functions becomes unmanageable very quickly as n increases.

This thesis develops a new approach for computing the spectral classes where the spectral operations are performed by manipulating the truth tables rather than first moving to the spectral domain to manipulate the spectral coefficients. Additionally, a generic approach is developed for modeling these spectral operations within the functional domain. The results of this research match previous for $n \leq 4$ but differ when $n = 5$ is considered. This research indicates with a high level of confidence that there are in fact 15 previously unidentified classes, for a total of 206 spectral classes needed to represent all 2^{2^n} Boolean functions.

Table of Contents

List of Tables

List of Figures

Science, my lad, is made up of mistakes, but they are mistakes which it is useful to make, because they lead little by little to the truth.

- Jules Verne

Chapter
1 - Introduction

1.0 Digital Logic And Boolean Switching Functions

In the field of digital logic, there are many techniques for representing circuits. One representation is shown in Figure 1.

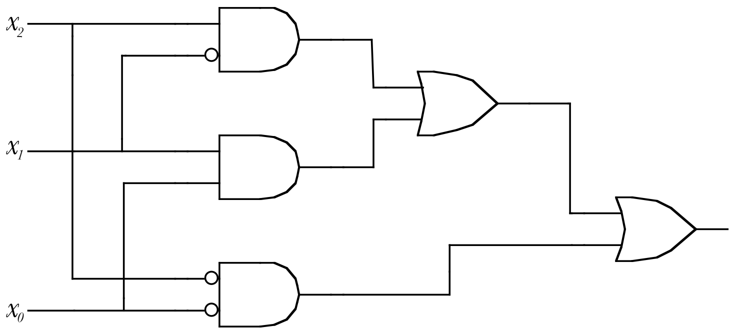

Figure 1 – Digital Circuit

Another representation is a truth table, which is used to tabulate the output for each possible combination of input values. For n inputs, a truth table lists 2^n possible results. For example, the circuit in Figure 1 can be represented by the truth table in Table 1.

x_2	x_1	x_0	
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Table 1 – Truth table for Figure 1

Circuits with even a moderate number of input variables generate truth tables that are very large. A more compact representation of the circuit can be achieved using a mathematical expression called a Boolean switching function. The circuit in Figure 1 can be represented by the Boolean switching function in Equation (1.1).

$$f(X) = x_2\overline{x}_2 + x_1x_0 + \overline{x}_2\overline{x}_0 \tag{1.1}$$

Often the output vector of the truth table may be encoded as an integer for an even more compact representation. In this case, the individual bits of the integer correspond to truth table values. For example, the truth table output vector for (1.1) is:

10111101

There are several ways of encoding this vector as an integer. In [2] the output vector of a function is encoded as an octal number. In octal, this function would be referred to as Function Number 275. It is possible to use any encoding scheme to represent a function, including hexadecimal and decimal. In this thesis, we have chosen to use a decimal representation. The decimal encoding for this example would be Function Number 189. As these encodings are simply different interpretations of the same underlying bit strings, the function numbers can be converted between representations for direct comparison.

As indicated by Hurst, *et al.* in [2], although encoding provides a compact representation of the functions, it does not "give any direct indication of functions of similar structure or complexity." A classification system could be used to group functions together based on other properties such as these, and also provide an even more compact representation of these functions.

1.1 Motivation

When considering Boolean switching functions with n variables, there are 2^{2^n} possible functions, each of which can be realized by enumerating all possible combinations of input values and arrangements of output values. As is expected with double exponential growth, the number of functions becomes unmanageable very quickly as n increases. To put this growth into perspective, $n=4$ produces a manageable 65,536 functions, while $n=6$ produces in excess of 1.8×10^{19} functions.

If one was to examine all possibilities for equivalent circuits when designing a processor, for example, it becomes impossible to consider all possible scenarios as the number of inputs for the circuit increases. Not all circuit realizations are considered equal, as some will provide superior qualities for power consumptions, physical space, latency, etc. Classification could assist this kind of application by making the navigation and selection of functions more manageable.

Using classification, all 2^{2^n} functions can be considered through a small number of representative functions. Hurst, *et al.* [2] list two advantages of classification:

1. *Increased understanding of functions that have essentially identical circuit realisations, leading to the classification of all* 2^{2^n} *functions of* $\leq n$ *variables in some compact manner.*

2. *Possibility of establishing a small set of "standard functions" or "prototype functions," from which any particular function may be realised by implementation of appropriate operations corresponding to the classification procedure.*

The spectral classes for functions with $n\leq5$ input variables were calculated in [2] and [4]. Considering 30 years have passed since the previous work was computed, it seemed reasonable that advances in computer hardware might allow for computing functions where $n>5$. With this possibility in mind, this research aims to reproduce the results from [2] and [4], albeit with a different approach, and attempts to harness current technology to calculate spectral classes for $n > 5$. The goals for this research are as follows:

1) Develop a new approach for computing spectral classes, and implement this approach.
2) Independently reproduce and verify the results published in [2], the spectral classes for $n = 5$, ensuring they are a valid basis for future work. This goal is to be carried out using the results of goal 1.
3) Use the knowledge gained in goal 2 to investigate the possibility of computing the spectral classes for functions with values of n greater than 5. If it is feasible to compute the spectral classes for $n > 5$, then provide the classes for as many values of n as possible.

1.2 Overview

In Chapter 2, this thesis provides an overview of various established Boolean function classification techniques. Chapter 2 also provides a more in-depth look at the established details of a particular classification technique within the spectral domain, as this is the focus of this thesis. Problems similar to spectral classification, and the approach used for previous work, are discussed in Chapter 3. In Chapter 4 various approaches for computing spectral classes are considered and discussed, including their advantages and disadvantages, with a focus on the approach used in this thesis. The new results from this research and the analysis of the approach used are presented in Chapter 5. Chapter 6 covers potential future work and improvements to the approach.

The implementation of the techniques discussed in this thesis are discussed in detail in Appendix A, including specific techniques used to optimize for execution time and resource usage. Included in Appendix B is the classification lists created by this implementation in Appendix A, and the transcription and reconstruction of the results from previous work. Finally, Appendix C contains the C++ source code that produced the results discussed in this thesis.

1.3 Summary

This research attempts to reproduce the spectral classes produced in [2] and [4] for $n \leq 5$ by performing all operations within the functional domain, rather than the spectral domain. Although the goal is spectral classification, the operations can be accomplished by manipulating the truth tables rather than first moving to the spectral domain to manipulate the spectral coefficients. The results of this research match [2] and [4] for $n \leq 4$ but differ when $n = 5$ is considered. This research indicates with a high level of confidence that there are in fact 15 additional classes, for a total of 206 spectral classes needed to represent all 2^{2^5} Boolean functions.

Computer hardware has not advanced enough for spectral classification of functions with $n > 5$ input variables to be calculated with existing approaches to classification, even with heavy optimization.

Chapter 2 - Background

2.0 Introduction

This chapter presents the concepts and definitions required for the topics covered in this thesis. Concepts introduced in this chapter include Boolean switching functions, classification, and the spectral domain. Additionally, we introduce a concept that we have termed "rules," which is essential to the classification approach introduced in this thesis.

2.1 Boolean Switching Functions

According to Hurst, *et al.* [2] and Rice [1], a Boolean switching function (referred to simply as a function for the remainder of this thesis) is a mathematical equation that describes a logic system based on Boolean logic operations. The basic logic operations are: AND, OR, NOT and XOR (exclusive-OR). There are also the operations NAND (not-AND), NOR (not-OR), and XNOR (not-exclusive-OR) which can be derived by combining the basic logic operations.

x_1	x_0	
0	0	0
0	1	0
1	0	0
1	1	1

(a)

x_1	x_0	
0	0	0
0	1	1
1	0	1
1	1	1

(b)

x_1	x_0	
0	0	0
0	1	1
1	0	1
1	1	0

(c)

$\overline{x}$	
0	1
1	0

(d)

Figure 2 – a) AND b) OR c) XOR d) NOT

Traditionally in Boolean logic, AND and OR are represented by the intersection ($\cap$), and union ($\cup$) symbols, but may also be represented by sum ($+$) and product ($\times$, or simply adjacent terms) symbols. For example, expression (2.1) represents an AND operation between variables x_0 and x_1, while expression (2.2) represents the OR operation. For the purposes of this thesis, the sum and product operators will be used.

$$x_0 x_1 \tag{2.1}$$

$$x_0 + x_1 \tag{2.2}$$

The exclusive-OR operator is represented by $\oplus$, while the NOT operator, also known as invert, negate, or bar, is traditionally represented by the symbol $\neg$ preceding the variable, or a solid bar above the variable as seen in equation (2.3).

$$\overline{x} \tag{2.3}$$

The output for the AND operation, as seen in Figure 2a, is true when the values of all input variables are true, and false in all other cases. For the OR operation, as seen in Figure 2b, the output is true when the value of at least one input variable is set to true. The output for the NOT operation, as seen in Figure 2d, is the opposite value of the input variable. Exclusive-OR, as seen in Figure 2c, has the output of true when there are an odd number of input variables with the value set to true, but false for all other cases, including when both input bits are set to true. The operations NAND, NOR and XNOR are simply the NOT operation applied to the output of AND, OR and XOR respectively.

2.2 Binary Representation

The decimal representation of a Boolean function relies on the assumption of a certain order of the output vector bits in the truth table. For this thesis, it is assumed that the most significant bit of the integer is the first row of an ordered truth table (the "zero" row). For example, as seen in Figure 3, the order of the bits to be encoded as an integer would be $abcdefgh$. If the output vector happens to be shorter than the data type used to represent it, the number is padded with zeros on the left hand side (most significant bits).

x_2	x_1	x_0	
0	0	0	**a**
0	0	1	**b**
0	1	0	**c**
0	1	1	**d**
1	0	0	**e**
1	0	1	**f**
1	1	0	**g**
1	1	1	**h**

Figure 3 – Truth table format

The convention used in this thesis is as follows: the input variables of a truth table are read from right to left. In the example in Figure 3, the first column on the right is the output vector, followed by the column for variable z and followed by variables y and x. This thesis also labels variables with a single letter and an incrementing subscript; the first variable on the right-most variable column can also be called input variable 1, or x_i and the left most column of the truth table is the nth input variable, or x_n. In general, when subscripts are not used, the term "variable y" would be used rather than "input variable 2" in that example.

Latin letters are used for output vector and truth table results when referring to generic cases rather than specific cases with defined truth values. In general, the letters chosen for these generic examples begin with lowercase a and increase incrementally.

2.3 Classification

Given that 2^{2^n} unique functions can be realized for n input variables, it is evident that even for relatively small values of n it is impossible to evaluate all possible functions. Once a classification technique is determined, if one considers a representative function of a given class as a generic black box circuit, it could be used as a building block for all functions within that class. To build a different circuit within this class, one would start with the black box circuit, and modify the inputs and/or outputs using simple operations equivalent to the established classification criteria. For example, a function may only differ from the representative function by an inverter on the output. With a generic circuit for each class, a potential application for classification is automatic circuit optimization (for power efficiency, physical space, speed, etc.) for an arbitrary value of *n*.

2.3.1 Definition Of Classification

The classification of a set of functions F into classes $Q_1, Q_2, \ldots, Q_p$ based on transformations $T_1, T_2, \ldots, T_m$ is such that:

$$F = Q_1 \cup Q_2 \cup \ldots \cup Q_p \qquad \text{and}$$
$$Q_i \cap Q_j = \varnothing \qquad \text{where } i \neq j$$

Two functions f_i, f_j, $i \neq j$, are in the same class Q_k if and only if f_i can be obtained from f_j by the application of some appropriate set of transformations from $T_1, \ldots, T_m$. No set of transformations applied to a function in Q_i can lead to a function in Q_j, for any $i, j \in \{1, \ldots, p\}$ where $i \neq j$.

2.3.2 Algebraic Classification

The most common classification scheme is based on NPN: Negation of input variables (N), Permutation of input variables (P), and Negation of output (N). This technique is referred to as algebraic classification as described by [4]. These transformations are described in detail in sections 2.3.2.2, 2.3.2.1, and 2.3.2.3.

2.3.2.1 Permutation Of Input Variables

As discussed previously, in the functional domain, a function can be represented as a truth table with columns for input variables on the left and a column for the output of the function on the right, as in Figure 4.

x_2	x_1	x_0	
0	0	0	**a**
0	0	1	**b**
0	1	0	**c**
0	1	1	**d**
1	0	0	**e**
1	0	1	**f**
1	1	0	**g**
1	1	1	**h**

Original

$\rightarrow$

x_1	x_2	x_0	
0	0	0	**a**
0	0	1	**b**
1	0	0	**c**
1	0	1	**d**
0	1	0	**e**
0	1	1	**f**
1	1	0	**g**
1	1	1	**h**

Permute x_2 and x_1

$\rightarrow$

x_1	x_2	x_0	
0	0	0	**a**
0	0	1	**b**
0	1	0	**e**
0	1	1	**f**
1	0	0	**c**
1	0	1	**d**
1	1	0	**g**
1	1	1	**h**

Sort

Figure 4 – Type 1: Permute input variables x_2 and x_1

To permute the input variables, the entire column of each variable to be affected is moved to its new location, while leaving the output vector untouched. The resulting table is no longer an ordered truth table (with input values incrementing from 0 through n). We then sort the rows to restore the ordered truth table by swapping entire rows. Sorting the truth table results in a new output vector, which represents the output vector of the new function. Using a bit string to represent the function, output vector *abcdefgh* is transformed into *abefcdgh* when permuting input variables x and y.

2.3.2.2 Negation Of Input Variables

To negate the input variables, the values in the entire column of each variable to be affected are inverted, while leaving the output vector untouched, as illustrated in Figure 5.

x_2	x_1	x_0	
0	0	0	**a**
0	0	1	**b**
0	1	0	**c**
0	1	1	**d**
1	0	0	**e**
1	0	1	**f**
1	1	0	**g**
1	1	1	**h**

15
Original

→

$\overline{x_2}$	x_1	x_0	
1	0	0	**a**
1	0	1	**b**
1	1	0	**c**
1	1	1	**d**
0	0	0	**e**
0	0	1	**f**
0	1	0	**g**
0	1	1	**h**

Negate x_2

→

$\overline{x_2}$	x_1	x_0	
0	0	0	**e**
0	0	1	**f**
0	1	0	**g**
0	1	1	**h**
1	0	0	**a**
1	0	1	**b**
1	1	0	**c**
1	1	1	**d**

Sort

Figure 5 – Type 2: Negate input variable x_2

The resulting table is no longer an ordered truth table (values incrementing from 0 to n). We then sort the rows to restore the ordered truth table layout by swapping entire rows. Sorting the truth table results in a new output vector, which represents the output vector of the new function. Using a bit string to represent the function, output vector *abcdefgh* is transformed into *efghabcd* when negating input variable x_2.

2.3.2.3 Negation Of Output

In the functional domain, the negation of the output is accomplished in a single step, as in Figure 6.

x_2	x_1	x_0	
0	0	0	**a**
0	0	1	**b**
0	1	0	**c**
0	1	1	**d**
1	0	0	**e**
1	0	1	**f**
1	1	0	**g**
1	1	1	**h**

Original

→

x_2	x_1	x_0	
0	0	0	$\neg$ **a**
0	0	1	$\neg$ **b**
1	1	0	$\neg$ **c**
1	1	1	$\neg$ **d**
1	0	0	$\neg$ **e**
1	0	1	$\neg$ **f**
0	1	0	$\neg$ **g**
0	1	1	$\neg$ **h**

Negate Output

Figure 6 – Type 3: Negate output vector values

To negate the output vector, the every value in the output vector is inverted.

2.3.3 Spectral Classification

In contrast to the output vector in the functional domain, which consists of the truth table output, a function is defined in the spectral domain by a vector of spectral coefficients.

The spectral domain has the distinct advantage of allowing one to see the "global picture of the network," rather than simply the "discrete nature of the data format." [2] As input variables are changed, one can see how it affects the entire function, making the spectral domain particularly useful.

The output vector in the spectral domain, S, consists of individual coefficients which use the notation s_α where subscript α is a subset of $1,\ldots,n$. Table 2 lists the "meaning" of the spectral coefficients, or how they correlate with the input variables of the function, for $n = 3$. This can be generalized for any value of n.

s_0	(number of false - number of true minterms)
s_1	similarity to input variable x_1
s_2	similarity to input variable x_2
s_{12}	similarity to input variable $x_1 \oplus x_2$
s_3	similarity to input variable x_3
s_{13}	similarity to input variable $x_1 \oplus x_3$
s_{23}	similarity to input variable $x_2 \oplus x_3$
s_{123}	similarity to input variable $x_1 \oplus x_2 \oplus x_3$

Table 2 - Correlation between spectral coefficients and input variables for $n = 3$

The generalized spectral coefficient output vector S for $n = 4$, as achieved by the Hadamard transformation matrix (further discussed in Section 2.3.3.1) can be observed in Figure 7.

$$s_0\ s_1\ s_2\ s_{12}\ s_3\ s_{13}\ s_{23}\ s_{123}\ s_4\ s_{14}\ s_{24}\ s_{124}\ s_{34}\ s_{134}\ s_{234}\ s_{1234}$$

Figure 7 – Spectral coefficients for $n = 4$

The coefficients are often reordered into groups according to their order, as seen in Figure 8, but the actual output order depends on the transformation matrix used in their calculation.

$$s_0;\ s_1 s_2 s_3 s_4;\ s_{12} s_{13} s_{14} s_{23} s_{24} s_{34};\ s_{123} s_{124} s_{134} s_{234};\ s_{1234}$$

Figure 8 – Reordered spectral coefficients for $n = 4$

The significance of the values are summed up by Hurst, Miller and Muzio [2] as follows:

i. *The sum of all spectral coefficients s of S for any fully defined function f(X) is* $\pm 2^n$
ii. *The maximum value of any individual s is* $\pm 2^n$*; this occurs when f(X) is identically equal to any row in* [the spectral transform matrix] *or its compliment. The range of each s is* $\{-2^n, -2^n+2, \ldots, 0, \ldots, 2^n-2, 2^n\}$
iii. *When any individual s is maximum-valued, all remaining* 2^n-1 *coefficients of S will be zero-valued*
iv. *When any input variable* x_i *is redundant in a given function f(X), the* 2^{n-1} *spectral coefficients that contain i in their subscript identification will all be zero-valued.*

Spectral classification encompasses the previously defined NPN classes and adds two additional operations involving the XOR Boolean operator. The NPN operation types 1 through 3 will be briefly defined in terms of the spectral domain while additional spectral operations (referred to as Type 4 and Type 5) will be described in terms of both functional domain and the spectral domain, as their functional domain properties have not previously been defined. All definitions are based on those by Hurst, Miller, and Muzio in [2].

In order to fully explain this technique, we must first provide some background on the spectral representation of a function, and how we calculate the spectral coefficients.

2.3.3.1 Computing Spectral Coefficients

Calculating the vector of spectral coefficients, S, is achieved with equation (2.4).

$$S = T^n Y \tag{2.4}$$

The T term is the Hadamard transformation matrix, which is defined in equation (2.5). The matrix is a complete $2^n \times 2^n$ matrix where n is the number of input variables. It is possible to use other transformation matrices, but the order of the spectral coefficients will be different. The Hadamard transformation matrix was chosen due to its recursive nature, and to allow direct comparison to previous work by other authors. The Hadamard transformation matrix's recursive definition makes it particularly attractive for programmatic implementation.

$$T^n \triangleq \begin{bmatrix} T^{n-1} & T^{n-1} \\ T^{n-1} & -T^{n-1} \end{bmatrix} \quad \text{where} \quad T^0 \triangleq +1 \tag{2.5}$$

The Y is the output vector of the function in the functional domain, but $+1/-1$ encoding is used rather than the more common $0/1$ encoding. Equation (2.6) can be used to convert the output vector Z (the $0/1$ encoded output vector of the truth table for the function) to the $+1/-1$ encoded Y vector.

$$y_j = 1 - 2z_j \quad \text{for all} \quad j \tag{2.6}$$

In equation (2.6), lower case y and z are considered to be elements of Y and Z respectively. An example of a Hadamard transformation matrix where $n = 3$ can be seen in the example in Figure 9.

$$\mathcal{T}^3 \triangleq \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix}$$

Figure 9 – Hadamard transformation matrix for $n = 3$

In Chapter 1, Function 189 that represents the circuit in Figure 1 is presented. Using the Hadamard transform, this function can be moved from the functional domain to the spectral domain.

x_2	x_1	x_0	
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Table 3 – Truth table for Function 189

The first step to transform Function 189 into the spectral domain is to take the output vector, $\mathcal{Z}$, from Table 3 and convert it to the +1/-1 encoding using (2.6). The conversion of vector $\mathcal{Z}$ to $\mathcal{Y}$ is demonstrated in Figure 10.

$$\begin{bmatrix}1\\0\\1\\1\\1\\1\\0\\1\end{bmatrix} \rightarrow \begin{bmatrix}1-2(1)\\1-2(0)\\1-2(1)\\1-2(1)\\1-2(1)\\1-2(1)\\1-2(0)\\1-2(1)\end{bmatrix} = \begin{bmatrix}-1\\+1\\-1\\-1\\-1\\-1\\+1\\-1\end{bmatrix}$$

Figure 10 – Example converting the output vector of Function 189 from Z to Y encoding

Using (2.4), the output vector Y can be transformed into the spectral domain as seen in Figure 11.

$$\begin{vmatrix}s_0\\s_1\\s_2\\s_{12}\\s_3\\s_{13}\\s_{23}\\s_{123}\end{vmatrix} = \begin{bmatrix}1&1&1&1&1&1&1&1\\1&-1&1&-1&1&-1&1&-1\\1&1&-1&-1&1&1&-1&-1\\1&-1&-1&1&1&-1&-1&1\\1&1&1&1&-1&-1&-1&-1\\1&-1&1&-1&-1&1&-1&1\\1&1&-1&-1&-1&-1&1&1\\1&-1&-1&1&-1&1&1&-1\end{bmatrix}\begin{bmatrix}-1\\+1\\-1\\-1\\-1\\-1\\+1\\-1\end{bmatrix} = \begin{vmatrix}-4\\0\\0\\-4\\0\\-4\\4\\0\end{vmatrix}$$

Figure 11 – Calculate the spectral coefficients for Function 189

The final output in the output vector, S, using the reordered format in Figure 8 is:

$$-4;\ \ 0\ 0\ 0;\ \ -4\ \ -4\ 4;\ \ 0$$

Note that the coefficients are "numerically equal to $\{\sum$ agreements between output $f(X)$ and the appropriate input function [minus] $\sum$ disagreements between $f(X)$ and the input function$\}$" [2].

2.3.3.2 Type 1: Permutation Of Input Variables

The spectral Type 1 operation, permutation of input variables, is the same operation as described in Section 2.3.1.2. We now discuss Type 1 in terms of the spectral domain rather than the functional domain.

In the spectral domain, if input variables x_i and x_j are swapped, all spectral coefficients which include subscripts i and j must also be swapped, as seen in (2.7).

$$\begin{gathered} s_i \leftrightarrow s_j \\ s_{ik} \leftrightarrow s_{jk} \\ s_{ikl} \leftrightarrow s_{jkl} \\ \vdots \end{gathered} \tag{2.7}$$

All other coefficients remain unchanged.

2.3.3.3 Type 2: Negation Of Input Variables

The spectral Type 2 operation, negation of input variables, is the same operation as described in Section 2.3.1.2. We now discuss Type 2 in terms of the spectral domain rather than the functional domain.

In the spectral domain, if an input variable x_i is negated, all spectral coefficients that include subscript i must be negated, as seen in (2.8).

$$\begin{gathered} s_i \leftrightarrow -s_i \\ s_{ij} \leftrightarrow -s_{ij} \\ s_{ikl} \leftrightarrow -s_{ikl} \\ \vdots \end{gathered} \tag{2.8}$$

All other coefficients remain unchanged.

2.3.3.4 Type 3: Negation Of Output

The spectral Type 3 operation, negation of output, is the same operation as described in section 2.3.1.3. We now discuss Type 3 in terms of the spectral domain rather than the functional domain.

In the spectral domain, if the output is negated, all spectral coefficients are negated, as seen in (2.9).

$$\begin{gathered} s_0 \leftrightarrow -s_0 \\ s_1 \leftrightarrow -s_1 \\ s_2 \leftrightarrow -s_2 \\ \vdots \\ s_{12\ldots n} \leftrightarrow -s_{12\ldots n} \end{gathered} \tag{2.9}$$

2.3.3.5 Type 4: Variable Replacement With XOR

In the functional domain, the input variables are represented in a truth table with the resulting output vector on the right hand side, as in Figure 12.

x_2	x_1	x_0	
0	0	0	**a**
0	0	1	**b**
0	1	0	**c**
0	1	1	**d**
1	0	0	**e**
1	0	1	**f**
1	1	0	**g**
1	1	1	**h**

Original

$\rightarrow$

$x_2 \oplus x_1$	x_1	x_0	
0	0	0	**a**
0	0	1	**b**
1	1	0	**c**
1	1	1	**d**
1	0	0	**e**
1	0	1	**f**
0	1	0	**g**
0	1	1	**h**

Substitute $x_2 \oplus x_1$ for x

$\rightarrow$

$x_2 \oplus x_1$	x_1	x_0	
0	0	0	**a**
0	0	1	**b**
0	1	0	**g**
0	1	1	**h**
1	0	0	**e**
1	0	1	**f**
1	1	0	**c**
1	1	1	**d**

Sort

Figure 12 – Type 4: Substitute $x_2 \oplus x_1$ for input variable x_2

To replace the input variables with an XOR expression, the entire column of each variable to be affected has its values replaced based on the result of the expression, while leaving the output vector untouched. The resulting table is no longer an ordered truth table (values incrementing from 0 to n). We sort the rows to restore the ordered truth table layout by swapping entire rows. Sorting the truth table results in a new output vector, which represents the output vector of the new functions.

In the spectral domain, if variable x_i is replaced with $x_i \oplus x_j$, all spectral coefficients which include subscripts i and ij must be swapped, as seen in (2.10).

$$\begin{gathered} s_i \leftrightarrow s_{ij} \\ s_{ik} \leftrightarrow s_{ijk} \\ s_{ikl} \leftrightarrow s_{ijkl} \\ \vdots \end{gathered} \tag{2.10}$$

All other coefficients remain unchanged.

2.3.3.6 Type 5: Output Replacement With XOR

In the functional domain, the input variables are represented in a truth table with the resulting output vector on the right hand side, as in Figure 13. To replace the output vector item with Type 5 modified values, the selected variable must be applied to each item in the output vector using the XOR operator. The resulting output vector is the new function created by the Type 5 classification transform.

x_2	x_1	x_0	
0	0	0	**a**
0	0	1	**b**
0	1	0	**c**
0	1	1	**d**
1	0	0	**e**
1	0	1	**f**
1	1	0	**g**
1	1	1	**h**

Original

→

x_2	x_1	x_0	
0	0	0	**a** $\oplus x_2$
0	0	1	**b** $\oplus x_2$
1	1	0	**c** $\oplus x_2$
1	1	1	**d** $\oplus x_2$
1	0	0	**e** $\oplus x_2$
1	0	1	**f** $\oplus x_2$
0	1	0	**g** $\oplus x_2$
0	1	1	**h** $\oplus x_2$

Output XORed by x_2

Figure 13 – Type 5: Perform XOR between the output vector and x_2

In the spectral domain the output of the function is replaced with the XOR of the function and a variable: $f(X) \rightarrow f(X) \oplus x_i$. As seen in (2.11), spectral coefficients with subscript i are swapped with its equivalent coefficient lacking an i subscript.

$$\begin{gathered} s_i \leftrightarrow s_0 \\ s_{ij} \leftrightarrow s_j \\ s_{ijk} \leftrightarrow s_{jk} \\ \vdots \end{gathered} \tag{2.11}$$

All pairs of spectral coefficients are swapped and therefore all 2^n coefficients are affected.

2.4 Spectral Signature

In [2] and [4], an abbreviated form is used to represent the properties of a spectral class. This thesis refers to this form as the function's spectral signature, or simply its signature. The signature's form is the made up of the spectral coefficient s_o, followed by the first order coefficients, and the "summary of the complete spectrum." [2] The summary of the complete spectrum is simply a list of the absolute values of the spectral coefficients and the number of occurrences of each. For example, using the function in Figure 11, the signature for Function 189 would be:

$$-4 \quad 0 \quad 0 \quad 0 \qquad 4\times 0 \quad 4\times 4$$

2.5 Other Function Groups

There have been numerous other approaches to grouping functions according to some specific property. Although the focus of this research is on spectral classification, it is worthwhile to consider these other approaches, as they have been the focus of classification attempts in the past.

2.5.1 Threshold

A threshold function, which may also be referred to as a linearly separable function, is a function where the 2^n minterms are represented as nodes in a n-dimensional space hyper-cube where a plane can "unambiguously divide all true ($f(X)=1$) nodes from all false ($f(X)=0$) nodes" [2]. The 2^n nodes are equally spaced. An example for $n=2$ can be seen in Figure 14.

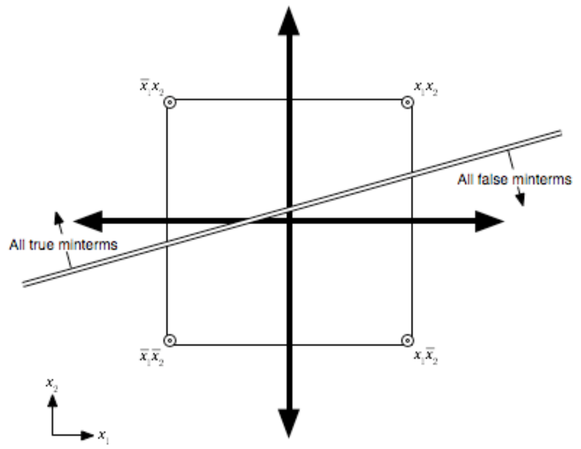

Figure 14 – Threshold function for $n = 2$

Although linear separation can be used to classify functions, the effectiveness of classification decreases as n increases. The equation of the plane can be calculated by equation (2.12), as given in [2].

$$a_1x_1 + a_2x_2 + ... + a_nx_n = d \quad \text{where all the } a_i \text{ are constants} \tag{2.12}$$

The axes of the n-dimensional hyper-cube are the input variables $x_1, x_2, ..., x_n$, while a_i and d are constants.

Given the equation of the plane, any node on the origin side of the partition will have equation (2.13).

$$a_1x_1 + a_2x_2 + ... + a_nx_n < d \tag{2.13}$$

The remaining nodes (on the plane and to the opposite side of the origin) have equation (2.14).

$$a_1x_1 + a_2x_2 + ... + a_nx_n \geq d \tag{2.14}$$

Therefore, one can say if a node has the equation (2.13) then $f(X) = 0$, otherwise if the node has equation (2.14) then $f(X) = 1$. Each constant a_i represents the "weight" or importance of an input variable to the output of the function in a threshold logic gate. [2]

2.5.2 Unate

A unate function is defined as a function where both a variable and its complement do not exist within a minimized sum of products representation of $f(X)$, according to [2]. For example, the function $f(X) = x_1\overline{x}_2 + x_1x_3$ is unate, while the function $g(X) = x_1\overline{x}_2 + \overline{x}_1x_3$ is not (both x_1 and $\overline{x}_1$ exist within a minimized sum of products expression for $g(X)$). Although this is a fairly simple criterion for classification, there exists a caveat that makes determining whether or not is unate very complex. A function must be minimized before it can be determined whether or not it is unate.

2.6 Rules

For the purposes of this thesis, a very specific definition for the term "rule" is used. A rule is the representation for the permutation of the output vector for a given transformation of the function. For example, in Rule (2.15), letters of the alphabet represent the original output vector of the function.

$$\{a,b,c,d,e,f,g,h\} \tag{2.15}$$

$$\{b,a,c,d,f,e,g,h\} \tag{2.16}$$

Rule (2.16) represents how the binary values of the original vector are interchanged. If the original output vector is $(0,1,1,0,1,0,1,0)$, then the rule stated in (2.16) states that the new output vector is $(1,0,1,0,0,1,1,0)$.

2.7 Double Exponential

For the purpose of this thesis, the term "double exponential," is used to describe a constant to the power of an exponential function, as seen in equations (2.17).

$$f(x) = a^{b^n} \tag{2.17}$$

Complexity analysis for double exponential algorithms results in $O(2^{c^n})$ where *a*, *b*, and *c* are constants.

2.8 Linear Independence

Let K be a commutative field. An ordered set of n elements $(x_1, x_2, \ldots, x_n)$, all $x_i \in K$ is an n-vector over K. The r n-vectors $y_1, y_2, \ldots, y_r$ are linearly dependent over K if there exist scalars $a_1, a_2, \ldots, a_r$ not all 0 such that:

$$a_1 y_1 + a_2 y_2 + \ldots + a_r y_r = 0$$

otherwise they are linearly independent. In our case, $K = \mathrm{GF}(2) = \{0,1\}$.

As an example, given the three vectors of $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$, it can be seen that they are linearly independent because no one vector can be created by combinations of the remaining vectors. In contrast, the vectors $(2,-1,1)$, $(1,0,1)$, and $(3,-1,2)$, are not linearly independent since the first two vectors can be added to create the third vector.

In this research, only the functional domain is considered for the constants a_i; in our case the constants $a_i \in \{0,1\}$. Note, the implementation of XOR is addition mod2. For each function the $2^n - 1$ equations in equation (2.18) must be calculated.

$$\begin{gathered}
0f_1 + 0f_2 + \cdots + 0f_{n-1} + 1f_n \neq 0 \\
0f_1 + 0f_2 + \cdots + 1f_{n-1} + 0f_n \neq 0 \\
\vdots \\
1f_1 + 1f_2 + \cdots + 1f_{n-1} + 0f_n \neq 0 \\
1f_1 + 1f_2 + \cdots + 1f_{n-1} + 1f_n \neq 0
\end{gathered} \tag{2.18}$$

2.9 The Problem

The number of functions for a given number of input variables increases double exponentially. This growth causes consideral problems when trying to process the functions. When considering all possible functions for a given number of input variables, n, as tabulated in Table 4, it becomes apparent how quickly the number of functions becomes unmanageable. For situations where $n < 5$, all functions can be considered and processed relatively easily. Even for $n = 4$, the total number of functions to consider is only 65,536, which would consume only 128KB of memory assuming the Boolean output vectors are stored as short integers (16 bits or 2 Bytes). Simply increasing n by 1 to $n = 5$ causes the number of functions to increase to over 4 million. To store all possible functions for $n = 5$, a full integer (32 bits or 4 Bytes) is needed, consuming over 16GB of memory. Increasing n again to $n = 6$, the number of functions pushes the memory usage to over 137,438,953,476GB of memory when storing the functions as long long integers (64 bits or 8 Bytes). Although today it is possible to load 16GB worth of data into primary storage, albeit not particularly feasible, it is currently not possible to load 137,438,953,476GB of data on any one storage device. Even without considering processing the data, the simple storage of all these functions quickly becomes impractical. Compromises could be made to not store all functions in memory at once, but at the cost of overall computing time as there is an increase in overhead due to memory management. The problem is already very computationally intensive, and adding the extra burden of memory swapping makes the approaches used in this implementation unfeasible.

n	Number of Functions	Times larger than $n = 5$
1	4	-
2	16	-
3	256	-
4	65,536	-
5	4,294,967,296	1
6	$1.8447744074 \times 10^{19}$	~4,294,967,296
7	$3.4028236692 \times 10^{38}$	$\sim 7.9 \times 10^{28}$
8	$1.1579208924 \times 10^{77}$	$\sim 2.7 \times 10^{67}$
9	$1.3407807930 \times 10^{154}$	$\sim 3.1 \times 10^{144}$
10	$1.7976931349 \times 10^{308}$	$\sim 4.2 \times 10^{298}$

Table 4 – The number of functions for $n = 1$ **through** 10

Even applying a small operation to all possible functions that would only take a single clock cycle in a processor would quickly become too slow to be useful.

As an example, consider a computer system where there is no overhead for loading data from memory, possesses a 2.5GHz processor, and which can perform 2,500,000,000 single clock cycle operations per second. If one were to use this system to process all possible functions for $n = 5$ with a single clock cycle operation, it would take approximately 1.7 seconds. Increasing n to $n = 6$, this same operation would now take 7,378,697,629 seconds, or nearly 234 years. This example is not realistic, as most useful operations would take many more than a single clock cycle to accomplish, and system overhead would be a fairly significant factor.

2.10 Summary

It is apparent that some form of classification is needed to abbreviate the list of functions that are processed, yet still be able to be assured that all possible functions are considered. Rather than iteratively trying every possible function looking for the most suitable function for a given purpose, the ability to process a certain subset of functions could make such a task feasible by reducing the number of functions significantly. Classification based on certain properties of the functions should accomplish this goal.

The concepts needed for this thesis, such as Boolean switching functions, classification, the spectral domain, and the newly introduced concept of rules are defined. These concepts are used and expanded upon in the subsequent chapters.

Chapter 3 - Related Work

3.0 Introduction

There are many problems that use the same techniques as spectral classification. Some problems are simply equivalent forms of spectral classification, while others are used for other types of classification.

The previous work of spectral classification in [2], which is the basis for this research, published only the list of spectral signatures, but lacks the required details needed for further analysis of the data. This thesis focuses on spectral coefficients that have been produced by the Hadamard transform, but it is by no means the only spectral transform available. Of the transforms available, some are canonical specializations of the Hadamard transform, while others expose entirely different properties of the function.

3.1 Alternate Representations Of The Hadamard Transform

In addition to spectral classification using the Hadamard transform, other transforms such as the Walsh, Rademacher-Walsh, and Walsh-Paley can be used, albeit with a different resulting spectral ordering. [2]

S_H	S_W	S_{RW}	S_{WP}
s_0	s_0	s_0	s_0
s_1	s_3	s_3	s_3
s_2	s_{23}	s_2	s_2
s_{12}	s_2	s_1	s_{23}
s_3	s_{12}	s_{23}	s_1
s_{13}	s_{123}	s_{13}	s_{13}
s_{23}	s_{13}	s_{12}	s_{12}
s_{123}	s_1	s_{123}	s_{123}

Figure 15 – Spectral output order for Hadamard (S_H), Walsh (S_W), Rademacher-Walsh (S_{RW}), and Walsh-Paley (S_{WP}) transforms

Although these transforms retain the same information as a Hadamard transform, they do not have a recursive definition that is particularly suitable for programmatic implementation. Additionally, the Hadamard spectral output maintains an ordering that compares to a standard truth table, as seen in Figure 15. In the logic design environment, it is common to use the term "Rademacher-Walsh transform," even if the implementation is based on another equivalent transform such as Hadamard.

3.2 Other Transforms

Transforms that are not equivalent to the Hadamard transform, such as the autocorrelation transform, can be used to examine other properties of classes of functions. The autocorrelation transform essentially compares the function to itself when shifted by a certain amount using an XOR. [1]

Transforms used for Reed-Muller and arithmetic expansion can also be useful, but these transforms are not further examined in this work, as they are not relevant to this thesis. Details on Reed-Muller and arithmetic expansion can be found in [8].

Not all transforms are discrete transforms. Other transforms such as the Haar [2][8] transform or the Fourier series of transforms can be used on non-Boolean and continuous data sets. The Fourier transform has both a continuous and discrete version. The discrete Fourier transform (DFT) is an approximation of the continuous Fourier transform. The DFT has a fast implementation, typically referred to as a fast Fourier transform (FFT), which is commonly used in spectral analysis, data compression, partial differential equations, and the multiplication of large integers or polynomials, to name a few.

The DFT and the Walsh transform are directly related, although the "kernel" for a DFT is more complex as it must support n possible values compared to the 2 possible values for a Walsh transform. However, the same FFT approach can be used for the rapid evaluation of the Walsh transform. These transforms are mentioned strictly for comparison purposes and will not be further discussed in this thesis; details of these transforms can be found in [6].

3.3 History Of The Problem

In the mid-1970s, Edwards classified all 2^{2^n} possible functions using the five spectral operations for $n \leq 5$ [7]. It was thought that the signature, as defined in Section 2.4, was sufficient to define the classes. In [7], it was reported that 47 spectral classes were required to completely classify all 2^{2^n} functions.

Further investigation in [4] discovered that one of the 47 classes published in [7] did not meet the definition of the classification. This offending class was therefore split into separate classes with the same signature in order fit the classification definition. This discovery also proved that the spectral signature on its own is not enough to classify all

2^{2^n} functions. The number of classes published in [4] increased to 48 classes over the original results published in [7] due to this split.

A complementary approach was taken in [2] using only the first 4 operation types to classify the 2^{2^n} functions. The approach used in [2] produced 191 classes, which were mapped to the equivalent classes in [4] using the signature of each class. As the classes in [2] have fewer criteria than the classes in [4], there are multiple classes defined in [2] for each class defined in [4].

The general process used in [7], [4], and [2] was to transform a starting function, f, into the spectral domain, S_f, and attempt to construct all other functions in the same spectral class using the five spectral transformations. In order for this to be computed in a reasonable amount of time for $n = 5$, the problem was extensively pruned and optimized. Unfortunately, the details of these optimizations are unavailable.

The goal of this research is to independently verify the results published in [2] using the first four spectral operations. As there is a history of errors with this problem, it seems necessary to check these results before building on them and attempting to compute the classes for larger values of n. To further contrast the original work, the processing in our approach takes place entirely in the functional domain, with the exception of the direct comparison to the previous work.

3.4 Summary

The techniques used in this research for spectral classification of Boolean functions relate to other problems such as the Fourier series and Reed-Muller codes. Additionally, there are transforms that can be used in place of the Hadamard transform used in this work.

These techniques are not new, and spectral classification for $n = 5$ has been previously been published. The published work, which serves as the basis for this research, unfortunately doesn't contain the necessary details needed to expand future work upon, and therefore is reproduced in this work.

Chapter 4 - Approaches

4.0 Introduction

There are two major approaches that can be used to perform spectral classification of Boolean functions. The most obvious approach for spectral classification is function manipulation within the spectral domain, as accomplished in [2] and [4]. The more counter-intuitive approach is to perform all of the equivalent transformations within the functional domain. Both approaches have their advantages and disadvantages, as discussed later in this chapter. This research chose the functional domain based on several of the advantages of this domain, as well as its uniqueness compared to the work in [2] and [4]. As the techniques used for function transformation within the functional domain are counter-intuitive, it is worthwhile discussing how to systematically generate all possible rules, and ensure that these transforms are applied to all possible functions.

4.1 The Spectral Domain

As seen in Chapter 3, previous classification was accomplished within the spectral domain. The functions are converted from the functional domain into the spectral domain, resulting in a spectrum of 2^n coefficients. The spectra are transformed using the four types of operations. These operations simply interchange or negate the 2^n spectral coefficients resulting in other functions that reside within the same spectral class.

One of the biggest advantages of the spectral domain is the global nature of the representation. Because of this global nature, it is possible to make observations about groups of functions and optimize accordingly.

The biggest disadvantage of this approach is the potential overhead of converting the functions into the spectral domain in the first place. This overhead can be reduced depending on the approach used to generate the functions. Additionally, there has been work on methods of moving between the functional and spectral domain quickly using a fast Hadamard transform [1][2][4]. Unlike the fast Hadamard transform, which is related to the fast Fourier transform, alternative methods of quickly transforming functions into the spectral domain have been proposed in [9] and [10] using decision diagrams.

4.2 The Functional Domain

We chose in this research to focus on the functional domain rather than first moving to the spectral domain to apply the transformation operations. In the functional domain, the operations are applied directly to the function's truth table representation. The operations change the order of the truth table, and therefore change the order of the bits in the output vector (with the exception of Type 3, which simply inverts the output values). All of the operations for a given value of n can be predetermined and represented as rules. These rules are simply templates that represent how the values of one function's output vector are interchanged to become another function. Any functions resulting from the transformations applied to the original reside within the same spectral class.

One of the biggest advantages of working within the functional domain is the elimination of conversion overhead when moving to the spectral domain. Although this overhead can be greatly reduced, as previously mentioned, it can quickly become impractical as the value of n increases.

Another advantage of working in the functional domain is that current computers are very good at low-level Boolean operations. Awareness of the details of the low-level execution of the high-level language code allows for improved overall speeds due to shorter, more efficient, machine code produced by the compiler. An example code that can assist the compiler in producing more efficient machine code can be found in A.3.2.2. As only Boolean values need to be stored, they can be stored very compactly within data structures such as Integers, and can be operated on with Boolean operations such as AND, OR, XOR, and SHIFT. All of these operations can be handled very quickly. [11]

The biggest disadvantage compared to the spectral domain is the local nature of the function. Since a single function cannot indicate anything about other functions, it is difficult to optimize. The only optimization that has been made in this work is to pre-filter the functions; to group the functions based on the number of true bits in the output vector of the function's truth table. This optimization can be made due to the nature of the Type 1, 2, and 4 operations where the number of true bits is not changed. As explained in Section 4.2.1, this optimization can also cover the Type 3 operations. Note that this optimization does not hold for Type 5 operations.

4.2.1 Pre-Filter

The functions are categorized according to the number of true minterms of the output vector, resulting in $2^n + 1$ categories. The trivial cases, 0 and 2^n only have 1 function each, leaving $2^n - 1$ non-trivial cases to consider.

Let function f have k true minterms so that f is in category C_k. Under a Type 3 transformation, a function $f \in C_k$ is translated into $f' \in C_{k'}$ where $k' = 2^n - k$.

Theorem

If $f_1 \in C_{k_1}$ and $f_2 \in C_{k_2}$, $k_1 \neq k_2$, $k_1, k_2 \leq 2^{n-1}$, then f_1 and f_2 are in different spectral classes.

Proof

In order to establish the result, it is necessary to prove that if $f_1 \in C_{k_1}$, then f_1 cannot be transformed into some function f_2, $f_2 \in C_{k_2}$, $k_1 \neq k_2$, $k_1, k_2 \leq 2^{n-1}$ by any of the four spectral transformations.

Consider each of the four possible transformations:

- Type 1: Clearly a permutation of the input variables cannot change the number of true minterms in the output vector.
- Type 2: Clearly a negation of an input variable cannot change the number of true minterms in the output vector.
- Type 3: The inverse of a function takes the number of true minterms outside the range considered.
- Type 4: The XOR of input variables is a more complex rearrangement of the input set; however, any rearrangement of the input set does not change the cardinality of the output vector.

□

The number of functions in each of these 2^n categories, C_k, for an n variable function can also be calculated using the binomial coefficient, $\binom{m}{r}$, where m is 2^n and r is the number of true minterms in that category.

The number of categories is reduced to $\frac{1}{2}(2^n)$ because functions with k and $2^n - k$ true minterms are combined into the same category as they are in the same spectral class because of the Type 3 transformation. Since the functions with k and $2^n - k$ true bits are combined into the same category, then:

$$\begin{aligned} |C_k| &= \binom{2^n}{k} + \binom{2^n}{2^n - k} \quad \text{where } k \neq 2^{n-1} \\ |C_k| &= \binom{2^n}{k} \quad \text{where } k = 2^{n-1} \end{aligned} \tag{4.1}$$

The example in Table 5, for $n = 3$ includes the trivial case where $k = 0$.

k		Functions
0, 8	$\binom{8}{0}+\binom{8}{8}$	2
1, 7	$\binom{8}{1}+\binom{8}{7}$	16
2, 6	$\binom{8}{2}+\binom{8}{6}$	56
3, 5	$\binom{8}{3}+\binom{8}{5}$	112
4	$\binom{8}{4}$	70
	2^{2^n}	256

Table 5 – Number of Functions for $n = 3$

If one considers only the first $2^{2^n - 1}$ functions, such as the implementation in Appendix A.3.1, the number of functions is also reduced to half for each category.

4.2.2 Operations

As previously defined, all of the functions within a spectral class can be realized from any one function and a combination of the specified four types of operations. The approach developed for this thesis is to create a list of rules (as defined in Section 2.6) for each type of the spectral operation. For each type of operation, a list of every possible outcome is created. The rules simply describe how the output bits from the current function are remapped to create a new function within the same spectral class.

4.2.2.1 Type 1: Permutation Of Input Variables

The Type 1 operation involves permuting the input variables of the function. It is possible to permute n input variables in $n!$ possible ways resulting in $n!$ rules, including the original function.

	Rules									Input Variables		
										x_2	x_1	x_0
0	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*		x_2	x_1	x_0
1	*a*	*c*	*b*	*d*	*e*	*g*	*f*	*h*		x_2	x_0	x_1
2	*a*	*b*	*e*	*f*	*c*	*d*	*g*	*h*		x_1	x_2	x_0
3	*a*	*c*	*e*	*g*	*b*	*d*	*f*	*h*		x_0	x_2	x_1
4	*a*	*e*	*b*	*f*	*c*	*g*	*d*	*h*		x_1	x_0	x_2
5	*a*	*e*	*c*	*g*	*b*	*f*	*d*	*h*		x_0	x_1	x_2

Table 6 – Type 1 Rules for $n = 3$

For $n = 3$, there are 6 Type 1 rules, as seen in Table 6. Recall that the Latin characters represent values of the individual bits in the output vector.

4.2.2.2 Type 2: Negation Of Input Variables

The Type 2 operation involves negating the input variables of the function. It is possible to negate n input variables in 2^n possible ways resulting in 2^n rules, including the original function.

	Rules									Input Variables		
										x_2	x_1	x_0
0	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*		x_2	x_1	x_0
1	*b*	*a*	*d*	*c*	*f*	*e*	*h*	*g*		x_2	x_1	$\neg x_0$
2	*c*	*d*	*a*	*b*	*g*	*h*	*e*	*f*		x_2	$\neg x_1$	x_0
3	*d*	*c*	*b*	*a*	*h*	*g*	*f*	*e*		x_2	$\neg x_1$	$\neg x_0$
4	*e*	*f*	*g*	*h*	*a*	*b*	*c*	*d*		$\neg x_2$	x_1	x_0
5	*f*	*e*	*h*	*g*	*b*	*a*	*d*	*c*		$\neg x_2$	x_1	$\neg x_0$
6	*g*	*h*	*e*	*f*	*c*	*d*	*a*	*b*		$\neg x_2$	$\neg x_1$	x_0
7	*h*	*g*	*f*	*e*	*d*	*c*	*b*	*a*		$\neg x_2$	$\neg x_1$	$\neg x_0$

Table 7 – Type 2 Rules for $n = 3$

Only the negation of the input variables is considered on the original ordering of the variables. For $n = 3$, there are 8 Type 2 rules, as seen in Table 7.

4.2.2.3 Type 3: Negation Of Output

The Type 3 operation involves negating the output vector of the function. There are 2 possible functions that can be realized based on this type of operation. The first function is the original unaltered function, while the second possible function is where the output bits of the original function are inverted. Based on the narrow definition of "rule" used in this research, a rule for Type 3 operation does not exist since there is no interchange of output bits.

0	a	b	c	d	e	f	g	h
1	$a \oplus 1$	$b \oplus 1$	$c \oplus 1$	$d \oplus 1$	$e \oplus 1$	$f \oplus 1$	$g \oplus 1$	$h \oplus 1$

Table 8 – Type 3 transformations for $n = 3$

The two Type 3 operations for $n = 3$ are listed in Table 8.

4.2.2.4 Type 4: Variable Replacement With XOR

The Type 4 operation involves replacing an input variable of a circuit with an XOR pre-filter such as the example in Figure 16. In the example, variable x_2 in Figure 16a is replaced with the resulting value of $x_2 \oplus x_0$, as seen in Figure 16b.

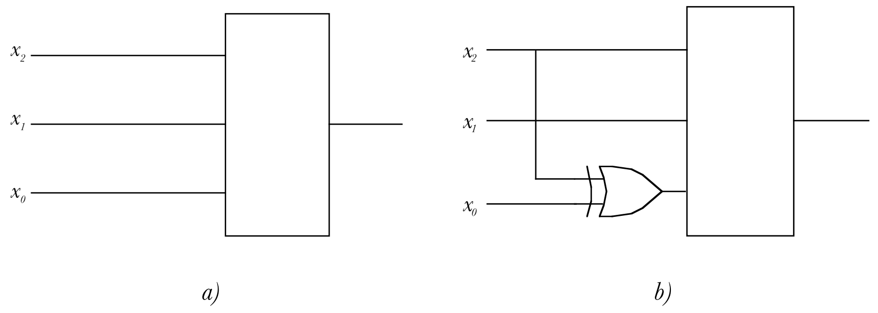

Figure 16 – a) Original function b) Function a with x_2 **replaced with** $x_2 \oplus x_0$

The variable replacement in this operation is limited, requiring that the original input variable, that is to be replaced, still be present in the final circuit. In other words, x_2 in Figure 16a may be replaced with $x_2 \oplus x_0$, but not with $x_1 \oplus x_0$, as x_2 is no longer part of the replacement. If the circuit is considered in the form of a matrix, where each row represents a variable, and the column indicates if the variable is included in the pre-filter replacement, circuits in Figure 16 would be represented by the matrices in Figure 17. In matrix form, each combination must have true bits on the diagonal.

$$\begin{array}{c} \\ x_2 \\ x_1 \\ x_0 \end{array}\begin{array}{c} \begin{array}{ccc} x_2 & x_1 & x_0 \end{array} \\ \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix} \end{array} \qquad \begin{array}{c} \\ x_2 \\ x_1 \\ x_0 \end{array}\begin{array}{c} \begin{array}{ccc} x_2 & x_1 & x_0 \end{array} \\ \begin{bmatrix} 1 & & 1 \\ & 1 & \\ & & 1 \end{bmatrix} \end{array}$$

a) *b)*

Figure 17 – True bits on the diagonal

A complete list of Type 4 operations must include all valid combinations of input variables with all valid pre-filter combinations. To produce this list, a list of all possible combinations of input variables, including both valid and invalid replacements, must be created. Within this list of all combinations the invalid combinations must be identified and removed leaving us with a list of all valid replacement input combinations. The technique used to removed invalid combinations is discussed later in this chapter.

To create the initial list of possible input replacements, a table of all possible replacements for a given variable is generated, as seen in Table 9. Each row of Table 9 contains a list of possible replacements for that particular input variable. Choosing one item from each row in Table 10 creates one possible Type 4 operation. This is repeated until all possible combinations have been realized. The list will contain $(2^{n-1})^n$ items.

x_3	$x_3 \oplus x_2$	$x_3 \oplus x_1$	$x_3 \oplus x_2 \oplus x_1$	$x_3 \oplus x_0$	$x_3 \oplus x_2 \oplus x_0$	$x_3 \oplus x_1 \oplus x_0$	$x_3 \oplus x_2 \oplus x_1 \oplus x_0$
x_2	$x_3 \oplus x_2$	$x_2 \oplus x_1$	$x_3 \oplus x_2 \oplus x_1$	$x_2 \oplus x_0$	$x_3 \oplus x_2 \oplus x_0$	$x_2 \oplus x_1 \oplus x_0$	$x_3 \oplus x_2 \oplus x_1 \oplus x_0$
x_1	$x_3 \oplus x_1$	$x_2 \oplus x_1$	$x_3 \oplus x_2 \oplus x_1$	$x_1 \oplus x_0$	$x_3 \oplus x_1 \oplus x_0$	$x_2 \oplus x_1 \oplus x_0$	$x_3 \oplus x_2 \oplus x_1 \oplus x_0$
x_0	$x_3 \oplus x_0$	$x_2 \oplus x_0$	$x_3 \oplus x_2 \oplus x_0$	$x_1 \oplus x_0$	$x_3 \oplus x_1 \oplus x_0$	$x_2 \oplus x_1 \oplus x_0$	$x_3 \oplus x_2 \oplus x_1 \oplus x_0$

Table 9 – Type 4 input variable lookup table for $n = 4$

Recall this is only a list of potential Type 4 operations, as some of these combinations are in fact invalid. To separate the valid operations from the invalid combinations, a linear independence check is done on the vectors of each input combinations (the rows of the matrix in Figure 17). If a combination is found to be linearly independent, it is added to the list of valid Type 4 operations.

		X_3	X_2	X_1	X_0
	0	x_3	x_2	x_1	x_0
8^4	1	x_3	x_2	x_1	$x_3 \oplus x_0$
	2	x_3	x_2	x_1	$x_2 \oplus x_0$
	…	…	…	…	…
	4094	$x_3 \oplus x_2 \oplus x_1 \oplus x_0$	$x_3 \oplus x_2 \oplus x_1 \oplus x_0$	$x_3 \oplus x_2 \oplus x_1 \oplus x_0$	$x_2 \oplus x_1 \oplus x_0$
	4095	$x_3 \oplus x_2 \oplus x_1 \oplus x_0$	$x_3 \oplus x_2 \oplus x_1 \oplus x_0$	$x_3 \oplus x_2 \oplus x_1 \oplus x_0$	$x_3 \oplus x_2 \oplus x_1 \oplus x_0$

Table 10 – All possible combinations of input variables from lookup table in Table 9

With a complete list of valid input variable replacements for Type 4 operations, each row can be used to generate the Type 4 rules using the method in Section 2.3.3.5. For $n = 3$, there are 34 Type 4 rules, as seen in Table 11. Only the XOR replacement of the input variables is considered on the original ordering of the variables.

4.2.2.5 Applying The Rules

Simply applying each of the rules for each rule type to a starting function will not achieve the desired result of producing all possible functions in a spectral class. The lists of rule only indicate the possible outcomes for that specific rule type, not combinations of all possible operations. In other words, applying all of the Type 1 rules to a function will only produce the functions resulting from combinations of permutations rather than combinations of any type of operation.

To create all possible functions, all of the rules must be considered in combination. In other words, one must combine all rules in all of the possible combinations. This concept can be best visualized by placing the rules in a tree, where each level of the tree represents a type of operation, as seen in Figure 18, where the dotted line represents the continuation of the pattern. The details of Figure 18 can be examined in Figure 19.

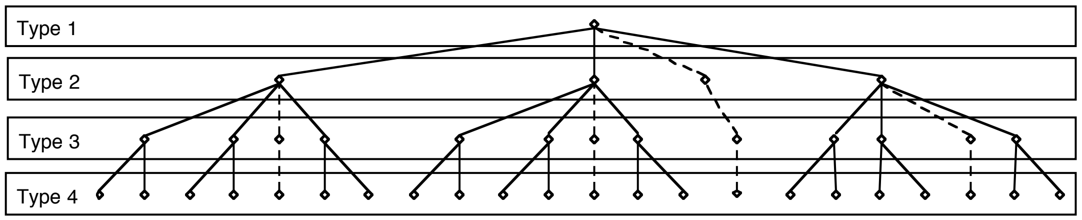

Figure 18 – Transformation rule combinations (Details in Figure 19)

Each result of the Type 1 rules must be operated on by each of the Type 2 rules. For each result of the Type 2 rules, the Type 3 operations must be applied. Finally, for each Type 3 result, each Type 4 operation must be applied. The leaf nodes of the tree will represent all possible functions that can be realized from the original starting function. The first leaf node, when considering a post-order traversal, represents the starting function: in other words, unmodified.

	Rules									Input Variables		
										x_2	x_1	x_0
0	a	b	c	d	e	f	g	h		x_2	x_1	x_0
1	a	b	c	d	f	e	h	g		x_2	x_1	$x_2 \oplus x_0$
2	a	b	d	c	e	h	h	g		x_2	x_1	$x_1 \oplus x_0$
3	a	b	d	c	f	e	g	h		x_2	x_1	$x_2 \oplus x_1 \oplus x_0$
4	a	b	c	d	g	h	e	f		x_2	$x_2 \oplus x_1$	x_0
5	a	b	c	d	h	g	f	e		x_2	$x_2 \oplus x_1$	$x_2 \oplus x_0$
6	a	b	d	c	h	g	e	f		x_2	$x_2 \oplus x_1$	$x_1 \oplus x_0$
7	a	b	d	c	g	h	f	e		x_2	$x_2 \oplus x_1$	$x_2 \oplus x_1 \oplus x_0$
8	a	d	c	b	e	h	g	f		x_2	$x_1 \oplus x_0$	x_0
9	a	d	c	b	h	e	f	g		x_2	$x_1 \oplus x_0$	$x_2 \oplus x_0$
10	a	d	c	b	g	f	e	h		x_2	$x_2 \oplus x_1 \oplus x_0$	x_0
11	a	d	c	b	f	g	h	e		x_2	$x_2 \oplus x_1 \oplus x_0$	$x_2 \oplus x_0$
12	a	b	g	h	e	f	c	d		$x_2 \oplus x_1$	x_1	x_0
13	a	b	h	g	f	e	c	d		$x_2 \oplus x_1$	x_1	$x_2 \oplus x_0$
14	a	b	h	g	e	f	d	c		$x_2 \oplus x_1$	x_1	$x_1 \oplus x_0$
15	a	b	g	h	f	e	d	c		$x_2 \oplus x_1$	x_1	$x_2 \oplus x_1 \oplus x_0$
16	a	h	g	b	e	d	c	f		$x_2 \oplus x_1$	$x_1 \oplus x_0$	x_0
17	a	h	g	b	d	e	f	c		$x_2 \oplus x_1$	$x_1 \oplus x_0$	$x_2 \oplus x_1 \oplus x_0$
18	a	g	h	b	f	d	c	e		$x_2 \oplus x_1$	$x_2 \oplus x_1 \oplus x_0$	$x_2 \oplus x_0$
19	a	g	h	b	d	f	e	c		$x_2 \oplus x_1$	$x_2 \oplus x_1 \oplus x_0$	$x_1 \oplus x_0$
20	a	f	c	h	e	b	g	d		$x_2 \oplus x_0$	x_1	x_0
21	a	f	h	c	e	b	d	g		$x_2 \oplus x_0$	x_1	$x_1 \oplus x_0$
22	a	h	c	f	g	b	e	d		$x_2 \oplus x_0$	$x_2 \oplus x_1$	x_0
23	a	h	f	c	g	b	d	e		$x_2 \oplus x_0$	$x_2 \oplus x_1$	$x_2 \oplus x_1 \oplus x_0$
24	a	h	c	f	e	d	g	b		$x_2 \oplus x_0$	$x_1 \oplus x_0$	x_0
25	a	h	f	c	d	e	g	b		$x_2 \oplus x_0$	$x_1 \oplus x_0$	$x_2 \oplus x_1 \oplus x_0$
26	a	f	c	h	g	d	e	b		$x_2 \oplus x_0$	$x_2 \oplus x_1 \oplus x_0$	x_0
27	a	f	h	c	d	g	e	b		$x_2 \oplus x_0$	$x_2 \oplus x_1 \oplus x_0$	$x_1 \oplus x_0$
28	a	f	g	d	e	b	c	h		$x_2 \oplus x_1 \oplus x_0$	x_1	x_0
29	a	f	d	g	e	b	h	c		$x_2 \oplus x_1 \oplus x_0$	x_1	$x_1 \oplus x_0$
30	a	g	f	d	h	b	c	e		$x_2 \oplus x_1 \oplus x_0$	$x_2 \oplus x_1$	$x_2 \oplus x_0$
31	a	g	f	d	h	b	c	e		$x_2 \oplus x_1 \oplus x_0$	$x_2 \oplus x_1$	$x_1 \oplus x_0$
32	a	d	g	f	e	h	c	b		$x_2 \oplus x_1 \oplus x_0$	$x_1 \oplus x_0$	x_0
33	a	d	g	f	h	e	c	b		$x_2 \oplus x_1 \oplus x_0$	$x_1 \oplus x_0$	$x_2 \oplus x_0$

Table 11 – Type 4 Rules for $n = 3$

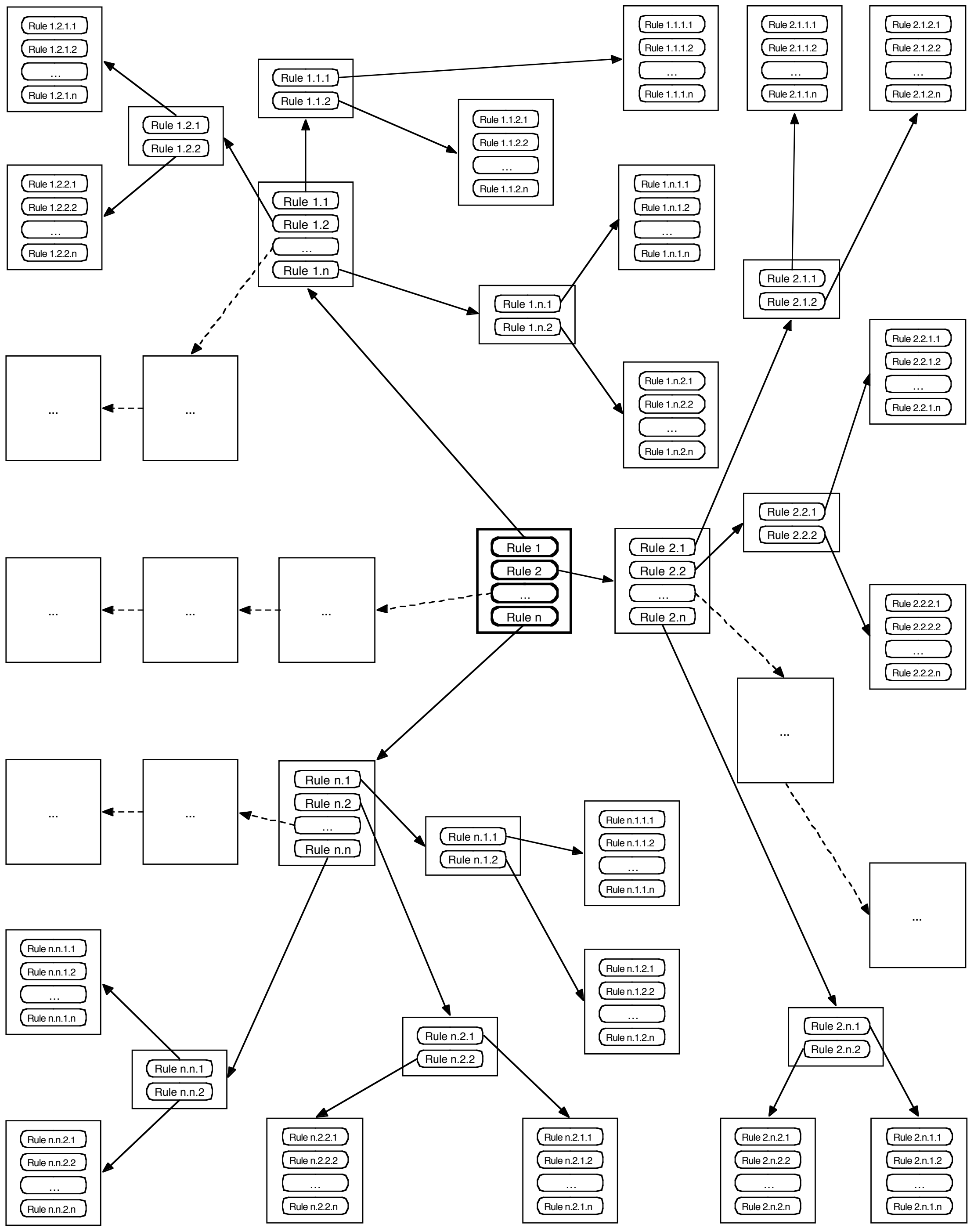

Figure 19 - Details of Figure 18

4.2.3 The Canonical Function

The function that we have chosen as the canonical function for each class is the first occurrence of that particular class number. The canonical function is defined to be the function whose representation is the smallest decimal integer in the class. This choice was made because the implementation used for the results published in this thesis encounters the functions in ascending order from Function 0 to 2^{2^n}.

In section 2.2, we noted that the decimal integer represents the output vector of the function's truth table where the bit of the zero row is the most significant bit, and the bit of the 2^n th row is the least significant. Due to this representation, the canonical function is the function that has its true bits concentrated towards the right hand side of the integer, or the 2^n th row of the truth table.

There do not appear to be any observations that can be made about the class based on the canonical function as it is in the functional domain. As previously stated, the functional domain can only give a "local view" of the function [1].

This representation is chosen because it is the simplest way to retrieve results based on the implementation. This may not be the best representation for the entire class, but because of the local view of the functional domain, it is not clear whether there is enough information to determine if any one function can best represent an entire class.

4.3 Summary

In order to spectrally classify Boolean functions, the functions must be either transformed and manipulated in the spectral domain, or in the less conventional functional domain. Previous work focused on the spectral domain, which is the most obvious approach for classifying functions spectrally.

This research focuses on working in the functional domain, which seems counter-intuitive if the functions are to be classified based on properties of the spectral domain. A systematic approach is proposed for generating all possible transforms and how to apply them to all possible functions, ensuring that all realizations are considered.

Chapter 5 - Results And Analysis

5.0 Introduction

In [2] it was stated that there are 191 spectral classes to represent all functions for $n = 5$. The work resulting in this thesis, however, indicates that several classes may inadvertently have been combined and that there are in fact 206 spectral classes needed to represent all 2^{2^5} functions. As classifying all functions for $n = 5$ is a problem that grows double-exponentially as the value of n increases, this problem must be optimized in order to compute a solution, and a computational solution is not easily checked. There is strong evidence indicating the results reported in this thesis are correct.

Careful analysis of the optimizations in this implementation indicates that 5.84×10^{11} transformations are required to classify all 2^{2^5} functions. Although this is still a lot of computation, it is significantly less than the 1.22×10^{19} transformations that must be performed when no optimization is added to the problem.

5.1 The New Results

The spectral classification of functions for $n \leq 5$ has been previously computed on several occasions. For both $n = 3$ and $n = 4$, this research agrees with the previous research that the number of classes is 6 and 18 respectively. For $n = 5$, the number of classes we generate differs from previous work. The classification of [2] list spectral coefficients for 191 classes, while this research finds 206 classes are necessary to represent all 2^{2^5} functions. The functions in Table 12 are the canonical functions for the extra 15 classes not present within the previous lists. Although the new research identifies 15 previously unidentified classes, the remaining 191 classes have a 1-to-1 pairing with the spectral classes listed in [2].

Class Number	Group Number	Function Number	s_0	s_1	s_2	s_3	s_4	s_5	Summary Of Complete Spectrum				
57	11	202079	10	10	6	10	10	14	1x14	5x10	7x6	19x2	
76	12	218479	8	8	4	8	8	16	1x16	8x8	16x4	7x0	
82	12	471895	8	8	12	8	8	12	2x12	8x8	14x4	8x0	
111	13	1514327	6	10	10	10	10	10	6x10	10x6	16x2		
124	14	472951	4	8	12	8	8	16	1x16	1x12	6x8	15x4	9x0
137	14	1514365	4	8	8	8	12	12	2x12	8x8	14x4	8x0	
138	14	1515325	4	8	8	12	8	12	2x12	8x8	14x4	8x0	
160	15	1515327	2	6	10	14	10	14	2x14	2x10	10x6	18x2	
163	15	1523007	2	6	10	10	10	14	1x14	5x10	7x6	19x2	
165	15	1523070	2	6	6	6	10	14	1x14	3x10	13x6	15x2	
170	15	18291671	2	10	10	10	10	10	6x10	10x6	16x2		
190	16	1523581	0	8	8	8	8	16	1x16	12x8	19x0		
191	16	1523582	0	4	8	8	8	16	1x16	8x8	16x4	7x0	
199	16	18291679	0	8	8	12	12	12	3x12	6x8	13x4	10x0	
200	16	18291709	0	8	8	8	12	12	2x12	8x8	14x4	8x0	

Table 12 – Previously unidentified classes for $n = 5$

All of the functions in Table 12 have a spectral signature that match classes contained within the other 191 classes. Much like the findings in [4], it is likely that the optimizations used in the implementation lead to the inadvertent combination of multiple classes.

All previous work [2][4] has taken place in the spectral domain, while this research demonstrates that spectral classification is viable in the functional domain. This new approach is conceptually similar to how these operations are carried out on hardware circuits. If one considers a generic black-box circuit, the operations manipulate the input and output of the circuit, and record the changes in the truth table. The implementation of this approach manipulates the truth tables of a circuit rather than moving the functional representation into the spectral domain, then perform the operations. This approach is unique because it proposes generic rules to describe the operations. The rules approach models these input and output manipulations at the truth table level as a generic description of the operation that is valid for all functions for a specified value of

n. A method is also proposed for combining the rule sets for all operations considered so that ensures that all 2^{2^n} functions are considered. These rules must be recalculated for each unique value of n, but the algorithm is the same for all values.

This thesis also proposes that algorithm for applying all combinations of all operations should not be optimized to ensure that all 2^{2^n} functions are considered. The alternative optimization that proposed is to prune the data set to reduce the overall work needed to compute the spectral classes. As is discussed later in this chapter, pruning the data set provides the most substantial reduction in overall work needed for this problem.

5.2 The Difficulties Of $n \geq 5$

When comparing this work to previous work [2][4], there is no discrepancy for the number of classes for $n < 5$. So what makes calculating and analysing for $n > 4$ so difficult?

The number of total functions grows double exponentially while the number of transformations that must be applied to these functions also grows very quickly. This means that while calculating for $n \leq 4$ can be accomplished quite easily, $n = 5$ is still a very large problem and prohibitively time consuming to compute in an unoptimized brute force approach.

One of the problems with optimizing to calculate the number of classes for 2^{2^5} functions is that since there are too many functions for a brute force approach, it is possible for one of the optimizations to lead to an error in the number of classes. The large number of functions also means it is impossible to check each of the results for correctness.

Using the same optimizations to calculate the classes for $n \leq 4$ can help indicate if there are errors in the optimizations, but it is not absolute proof. As seen when comparing this research to [2] and [4], the number of classes only disagree when $n > 4$. There are a few factors that may indicate why this might be.

It seems as though the coefficients "behave" differently for $n > 4$. As the value of n increases, there are more ways to combine the truth values of each function. It is possible that that not all combinations of the four operation types are needed to compute all of the classes for $n \leq 4$. At one time, it was thought that the "signature", which consists of the zero and first order coefficients plus a count of the coefficient magnitudes, was unique to a class and that classification could be done in this way. For $n \leq 4$, these signatures are unique, but once $n = 5$ is considered, the signature for each class is no longer unique.

It is possible that the spectral classes for $n < 5$ are in fact special cases and it is not possible to observe any patterns without knowing the classes for $n > 5$. Once spectral classes for $n > 5$ are calculated, it may be possible that there are different patterns for even and odd values of n. Unfortunately, until the spectral class structure for higher values of n are computed, the answers to these questions will remain unknown.

The list of spectral classes listed in [2] and [4] notes that previous work has determined that Hurst class 45 should in fact be split into two different classes, and that the defined signature is not enough to differentiate the functions. All of the new functions in Table 12 share signatures with previously defined spectral classes.

The biggest difficulty in comparing current work to past work is the lack of a complete set of previous test results. Although a summary of each canonical function has been listed in [2], it is impossible to determine the exact canonical function based on this information. Due to the age of the original research, much of the raw data has also been lost. Much of the comparison between current work and previous work is accomplished by comparing the results in [2] with some intermediate raw data listed in section B.2. Additionally, some assumptions have to be made when comparing current results to the list in [2] as there are times when it is impossible to differentiate between classes that contain the same signature.

Source code, or a list of optimizations to the original work is also not available. Without knowing what approach was used, or how the problem set was reduced, it is difficult to determine exactly why there is a discrepancy between results.

5.3 Evidence

Although we were unable to check each individual function to confirm they are in the appropriate spectral class, a number of approaches were used to instil confidence in these results. This seems necessary given the discrepancies with the previous research, and the inability to directly verify the results of the previous work.

Any of the individual tests, described in the following sections, are not enough in themselves to indicate likely correctness. Collectively these tests show that it is less likely that the results are incorrect, as it would take many compounded errors and coincidences to have incorrect data, yet pass all of these tests. If this work is to be used as the basis for future work, it is important that the confidence in these results is as high as possible. In addition to increasing the confidence in these results, these tests provide insight into the optimizations, and the avenues for future work.

5.3.1 Compared Functions

Each canonical function was converted to the spectral domain and the signatures were compared to the signatures in [2]. As there is no complete spectral listing for the classes in [2], it was not possible to know the exact function we were comparing against. It is, however, possible to compare the magnitude counts between the lists. It turns out that there is a 1-to-1 match between the 191 classes in the list in [2] and 191 of the classes of this research. In this research, there are an additional 15 classes that share magnitude counts, possibly indicating classes that have previously been inadvertently combined.

5.3.2 Number Of Coefficients Per Class And Group

The approach used in this work to generate the spectral classes uses a pre-filter step that separates the functions into groups based on the number of true bits in the output vector, as described in Section 4.2.1. As stated in that section, it is also possible to calculate the number of functions in each such group.

A count for the number of functions within each class has been generated, and the numbers for the functions that reside in each pre-filter group are added up. The sum of the function within each group is equal to the calculated (and expected) number. The implementation only considers the first $2^{2^n - 1}$ functions; therefore the number of functions per group, and in total, is exactly half.

Pre-filter Group	Class Number	Number of Functions	Group Sum
1	0	1	1
2	1	8	8
3	2	28	28
4	3	56	56
5	4	7	
	5	28	35
=		128	128

Table 13 – Number of functions per class and group for $n = 3$

For $n = 3$, there are a total of 128 total functions in 5 pre-filter groups, as seen in Table 13.

Pre-filter Group	Class Number	Number of Functions	Group Sum
1	0	1	1
2	1	16	16
3	2	120	120
4	3	560	560
5	4	140	
	5	1,680	1,820
6	6	1,680	
	7	2,688	4,368
7	8	840	
	9	6,720	
	10	448	8,008
8	11	240	
	12	6,720	
	13	4,480	11,440
9	14	15	
	15	960	
	16	420	
	17	5,040	6,435
=		32,768	32,768

Table 14 – Number of functions per class and group for $n = 4$

Much like $n=3$, the total number of functions for $n=4$ is as calculated. There are 32,768 functions in 9 pre-filter groups for $n=4$ as seen in Table 14.

Pre-filter Groups	Class Number	Number of Functions	Group Sum
1	0	1	1
2	1	32	32
3	2	496	496
4	3	4,960	4,960
5	4	1,240	
	5	34,720	35,960
6	6	34,720	
	7	166,656	201,376
7	8	17,360	
	9	416,640	
	10	27,776	
	11	444,416	906,192
8	12	4,960	
	13	416,640	
	14	277,760	
	15	2,222,080	
	16	444,416	3,365,856
9	17	620	
	18	119,040	
	19	52,080	
	20	624,960	
	21	3,333,120	
	22	4,444,160	
	23	277,760	
	24	1,666,560	10,518,300
10	25	14,880	
	26	416,640	
	27	277,760	
	28	952,320	
	29	833,280	
	30	9,999,360	
	31	6,666,240	
	32	6,666,240	
	33	2,222,080	28,048,800
11	34	52,080	
	35	416,640	
	36	27,776	
	37	119,040	
	38	6,666,240	
	39	4,444,160	
	40	83,328	
	103	19,998,720	
	104	6,666,240	
	105	79,994,880	
	106	277,760	
	107	9,999,360	
	108	6,666,240	
	109	6,666,240	
	110	19,998,720	
	111	**53,329,920**	143,775,520
15	112	7,440	
	113	555,520	
	114	208,320	
	115	4,999,680	
	116	833,280	
	117	833,280	
	118	6,666,240	
	119	833,280	
	120	4,999,680	
	121	26,664,960	
	122	19,998,720	
	123	39,997,440	
	124	**26,664,960**	
	125	26,664,960	
	126	6,666,240	
	127	2,222,080	
	128	138,880	
	129	4,999,680	
	130	2,499,840	
	131	1,666,560	
	132	19,998,720	
	133	9,999,360	
	134	3,333,120	
	135	9,999,360	
	136	79,994,880	
	137	**39,997,440**	
	138	**79,994,880**	
	139	6,666,240	
	140	39,997,440	
	141	3,333,120	471,435,600
16	142	992	
	143	119,040	

Pre-filter Groups	Class Number	Number of Functions	Group Sum	Pre-filter Groups	Class Number	Number of Functions	Group Sum
	41	4,999,680			144	833,280	
	42	1,666,560			145	3,333,120	
	43	19,998,720			146	138,880	
	44	2,222,080			147	6,666,240	
	45	19,998,720			148	2,499,840	
	46	317,440			149	4,999,680	
	47	833,280			150	39,997,440	
	48	2,666,496	64,512,240		151	2,222,080	
12	49	104,160			152	6,666,240	
	50	166,656			153	6,666,240	
	51	833,280			154	2,499,840	
	52	6,666,240			155	9,999,360	
	53	444,416			156	6,666,240	
	54	1,666,560			157	6,666,240	
	55	9,999,360			158	6,666,240	
	56	13,332,480			159	39,997,440	
	57	**19,998,720**			**160**	**39,997,440**	
	58	19,998,720			161	79,994,880	
	59	317,440			162	26,664,960	
	60	6,666,240			**163**	**39,997,440**	
	61	9,999,360			164	79,994,880	
	62	31,997,952			**165**	**19,998,720**	
	63	166,656			166	6,666,240	
	64	6,666,240	129,024,480		167	53,329,920	
13	65	8,680			168	444,416	
	66	104,160			169	19,998,720	
	67	1,666,560			**170**	**31,997,952**	
	68	2,666,496			171	19,998,720	565,722,720
	69	208,320		17	172	31	
	70	1,249,920			173	7,936	
	71	9,999,360			174	29,760	
	72	26,664,960			175	416,640	
	73	3,333,120			176	277,760	
	74	3,333,120			177	4,999,680	
	75	3,333,120			178	4,444,160	
	76	**9,999,360**			179	4,340	
	77	2,222,080			180	833,280	
	78	4,999,680			181	156,240	
	79	39,997,440			182	624,960	
	80	39,997,440			183	9,999,360	
	81	3,333,120			184	277,760	
	82	**39,997,440**			185	1,666,560	
	83	26,664,960			186	416,640	
	84	13,888			187	2,499,840	
	85	833,280			188	26,664,960	
	86	2,499,840			189	39,997,440	
	87	2,666,496	225,792,840		**190**	**4,999,680**	
14	88	34,720			**191**	**9,999,360**	
	89	138,880			192	6,666,240	
	90	1,666,560			193	4,999,680	
	91	2,499,840			194	39,997,440	
	92	3,333,120			195	39,997,440	
	93	13,332,480			196	222,208	
	94	6,666,240			197	9,999,360	
	95	6,666,240			198	3,333,120	
	96	2,222,080			**199**	**26,664,960**	
	97	39,997,440			**200**	**39,997,440**	
	98	26,664,960			201	15,998,976	
	99	8,888,320			202	833,280	
	100	1,666,560			203	833,280	
	101	19,998,720			204	2,666,496	
	102	9,999,360			205	13,888	300,540,195
				=		2,147,438,648	2,147,438,648

Table 15 – Number of functions per class and group for $n = 5$

The statistics for $n = 5$ continues the trend with 2,147,438,648 functions in 17 pre-filter groups. The new spectral classes have been bolded in Table 15.

Examining Table 15, it is evident that all classes contain at least 1 function. The previously unidentified classes are classes that encapsulate a large number of functions, and therefore are not classes that are special cases, such as Class 0. Additionally, all of the previously unidentified classes have function counts that occur elsewhere in the list, indicating that these classes are not mistakes. For example, both Class 195 and Class 200 have the same number of functions.

5.3.3 Calculations Of The Number Of Rules

It is possible to calculate the number of rules required to transform a starting function into all other possible functions within that class. The number of rules for $n \leq 6$ can be seen in Table 16. These numbers are valuable for comparing to the number of rules generated by this implementation.

	n	$n=1$	$n=2$	$n=3$	$n=4$	$n=5$	$n=6$
Type 1	$n!$	1	2	6	24	120	720
Type 2	2^n	2	4	8	16	32	64
Type 4	n/a*	1	3	34	1688	370,752	347,638,784†

Table 16 – Number of rules for $n \leq 6$‡

For Type 1 transformations, $n!$ rules are needed to produce all possible permutation of the input variables while Type 2 transformations require 2^n rules. Type 3 transformations do not have rules, as defined in Section 2.6, but rather the output of the function is simply inverted. Therefore, for all values of n, the number of transformations for Type 3 is simply the constant 2. Type 4 transformations are a bit more complicated as there is no known general case to calculate the required number of rules. In general, only an upper bound can be specified, as in section 4.2.2.4, which includes invalid transformations.

* The generalized equation is currently unknown

† This value has not been confirmed with Maple

‡ Generally spectral classification is only considered for $n \geq 3$

Although the number of rules generated by this implementation for Types 1 and 2 are confirmed, Type 4 cannot be compared if no general case can be provided. To confirm the number of Type 4 rules, an alternate method of generating the rules was created and its results compared to the implementation described in Appendix A. Note that all rules for the spectral operations include the original non-permuted ordering.

The alternative method was created using Maple, and all possible combinations of input variables for each value of n, in the same method used for Table 10, were checked for linear independence. Each set of input variables was checked using the determinant function that is built into Maple.

$$\begin{vmatrix} V_0 \\ V_1 \\ \cdots \\ V_n \end{vmatrix}_{\bmod 2} \neq 0 \tag{5.1}$$

If the result of the determinant modulo 2 does not equal 0, then it is considered to be linearly independent, and therefore a valid combination of input variables.

For $n \le 5$, the number of linearly independent sets returned from Maple equalled the number of Type 4 rules generated by the implementation in Appendix A.

In this research, and attempt was made to identify a general case for calculating the number of Type 4 rules. This attempt at a generalized closed formula failed, and a search for a known sequence of integers was undertaken. The On-Line Encyclopedia Of Integer Sequences [6] was searched for an existing sequence which includes 34, 1688, 370752. Currently, there are no sequences that include 34, 1688, 370752 in [6].

5.4 Other Analysis

5.4.1 Complexity

Spectral classification of Boolean functions is a very large problem and the number of transformations that must take place can be described by Expression (5.2):

$$A \times B \times C \times D \times E \tag{5.2}$$

where:

A: The total number of functions to be considered

B: The number of Type 1 transformations

C: The number of Type 2 transformations

D: The number of Type 3 transformations

E: The upper bounds§ for Type 4 transformations

In the worst case, this is an upper bound. The implementation of this approach is highly dependant on the order of (5.2). Although (5.2) uses the product symbol, this expression is not commutative as expected with. In this implementation, for each item saved in *A*, the work associated with parts *B*, *C*, *D*, and *E* can be completely avoided. This is true for every term going from right to left in the expression. In other words, for each item saved in *B*, work in *C*, *D*, and *E* are avoided. For each item saved in *C*, work in *D*, and *E* are avoided, and so on.

Alternatively, consider this expression as a tree where Figure 18 represents the terms *B*, *C*, *D*, and *E*. In this tree, Figure 18 is the child node for each item in *A*. If there are 2^{2^n} starting functions with *A*, it means that Figure 18 must be traversed 2^{2^n} times; once for each item of *A*. The optimization approaches in this section are simply methods to prune

§ See section 5.3.3

this tree. The further up the tree these optimizations occur, the larger the overall benefit to the problem.

The analysis of (5.2) is first considered in its worst case, and then progressively refined as optimizations are introduced in the following sections. Optimizations are most effective when earlier components of (5.2) are avoided as subsequent terms are also avoided. Therefore, the focus of this research is to reduce the number of earlier terms that need to be considered.

5.4.1.1 Brute Force

In the worst case, the total number of function transformations that must be performed is:

$$A:\ 2^{2^n}$$

$$B:\ n!$$

$$C:\ 2^n$$

$$D:\ 2$$

$$E:\ (2^{n-1})^n$$

Therefore, there are $2^{2^n} \times n! \times 2^n \times 2 \times (2^{n-1})^n$ transformations that must be performed to spectrally classify all functions of n input variables.

5.4.1.2 Optimization

The number of starting functions can be reduced by half by observing that the second half of all 2^{2^n} is simply a Type 3 operation applied to the first 2^{2^n-1} functions.

Recall that the order in which the rules are applied according to expression (5.2) is important, and cannot be changed. The optimized implementation of this thesis begin with:

$$A{:}\ \ 2^{2^n-1}$$

$$B{:}\ \ n!$$

$$C{:}\ \ 2^n$$

$$D{:}\ \ 2$$

$$E{:}\ \ (2^{n-1})^n$$

Therefore there are $2^{2^n-1} \times n! \times 2^n \times 2 \times (2^{n-1})^n$ transformations in the optimized general case for functions with n input variables. Since all possible combinations of all four spectral transformations are applied to a starting function, all functions that exist within the same class as the starting function are also discovered. This observation further reduces the number of starting functions from 2^{2^n-1} to the number of spectral classes, S_n where n is the number of input variables. As a result of the second optimization to A, the number of transformations in the optimized general case is now $S_n \times n! \times 2^n \times 2 \times (2^{n-1})^n$.

For $n = 5$, the total number of transformations is:

$$A{:}\ \ 205$$

$$B{:}\ \ 5!$$

$$C{:}\ \ 2^5$$

$$D{:}\ \ 2$$

E: 370,752[**]

[**] As calculated in section 5.3.3

Therefore the number of spectral transformations required to classify all functions for $n = 5$ is 5.84×10^{11} which is significantly smaller than the brute force case that would require 1.22×10^{19} spectral transformations.

5.4.3 Prediction

It has already been shown that classification of all 2^{2^n} functions is an incredibly difficult problem, especially as n increases to values above 4. For values of n where $n > 5$, it is impractical to use current methods of classification; therefore some other method is needed to calculate these classes.

One approach is to use existing data from smaller values of n and extrapolating the data for the desired value of n. Using prediction of this nature, it may be possible to derive all, or a large portion, of the classes for $n+1$ variables based on the data from lesser values of n. This could greatly decrease the amount of processing needed and could make classification for $n > 5$ feasible.

When considering the first 128 functions for $n = 3,4,5$, as seen in Table 17, many of the classes remain the same as the value of n increases. In Table 17, class numbers that do not match for a given function, for all of $n = 3,4,5$, have been highlighted. So far, it is unclear whether any conclusions can be drawn from this data.

Another trend is evident when the list is split into sections containing 4 functions. For the majority of these groups, the second and third functions from these groups share values. There are a couple of exceptions near the middle of the table, which also makes the outcome of this approach unclear.

Function Number	Class			
	$n=3$	$n=4$	$n=5$	
0	0	0	0	
1	1	1	1	
2	1	1	1	
3	2	2	2	
4	1	1	1	
5	2	2	2	
6	2	2	2	
7	3	3	3	
8	1	1	1	
9	2	2	2	
10	2	2	2	
11	3	3	3	
12	2	2	2	
13	3	3	3	
14	3	3	3	
15	4	4	4	
16	1	1	1	
17	2	2	2	
18	2	2	2	
19	3	3	3	
20	2	2	2	
21	3	3	3	
22	3	3	3	
23	5	5	5	
24	2	2	2	
25	3	3	3	
26	3	3	3	
27	5	5	5	
28	3	3	3	
29	5	5	5	
30	5	5	5	
31	**3**	**6**	**6**	X
32	1	1	1	
33	2	2	2	
34	2	2	2	
35	3	3	3	
36	2	2	2	
37	3	3	3	
38	3	3	3	
39	5	5	5	
40	2	2	2	
41	3	3	3	
42	3	3	3	
43	5	5	5	
44	3	3	3	
45	5	5	5	
46	5	5	5	
47	**3**	**6**	**6**	X
48	2	2	2	
49	3	3	3	
50	3	3	3	
51	4	4	4	
52	3	3	3	
53	5	5	5	
54	5	5	5	
55	**3**	**6**	**6**	X
56	3	3	3	
57	5	5	5	
58	5	5	5	
59	**3**	**6**	**6**	X
60	4	4	4	
61	**3**	**6**	**6**	X
62	**3**	**6**	**6**	X
63	**2**	**8**	**8**	X
64	1	1	1	
65	2	2	2	
66	2	2	2	
67	3	3	3	
68	2	2	2	
69	3	3	3	
70	3	3	3	
71	5	5	5	
72	2	2	2	
73	3	3	3	
74	3	3	3	
75	5	5	5	
76	3	3	3	
77	5	5	5	
78	5	5	5	
79	**3**	**6**	**6**	X
80	2	2	2	
81	3	3	3	
82	3	3	3	
83	5	5	5	
84	3	3	3	
85	4	4	4	
86	5	5	5	
87	**3**	**6**	**6**	X
88	3	3	3	
89	5	5	5	
90	4	4	4	
91	**3**	**6**	**6**	X
92	5	5	5	
93	**3**	**6**	**6**	X
94	**3**	**6**	**6**	X
95	**2**	**8**	**8**	X
96	2	2	2	
97	3	3	3	
98	3	3	3	
99	5	5	5	
100	3	3	3	
101	5	5	5	
102	4	4	4	
103	**3**	**6**	**6**	X
104	3	3	3	
105	4	4	4	
106	5	5	5	
107	**3**	**6**	**6**	X
108	5	5	5	
109	**3**	**6**	**6**	X
110	**3**	**6**	**6**	X
111	**2**	**8**	**8**	X
112	3	3	3	
113	5	5	5	
114	5	5	5	
115	**3**	**6**	**6**	X
116	5	5	5	
117	**3**	**6**	**6**	X
118	**3**	**6**	**6**	X
119	**2**	**8**	**8**	X
120	5	5	5	
121	**3**	**6**	**6**	X
122	**3**	**6**	**6**	X
123	**2**	**8**	**8**	X
124	**3**	**6**	**6**	X
125	**2**	**8**	**8**	X
126	**2**	**8**	**8**	X
127	**1**	**11**	**12**	X

Table 17 – Comparison between classes for $n = 3,4,5$

5.5 Results Confidence

The data resulting from the work in this research differs from previously published data in [2] and [4]. The obvious question is whether this data can be considered more or less reliable than the data it is being compared to. It is still infeasible to verify each class for correctness, but there is very strong evidence that the new class structure is correct.

The approach used in this research considers nearly all of the 2^{2^n} possible functions rather than extensively pruning the problem. The pruning applied to the problem, as described in Section 5.4.1.2, is fairly straightforward and it is easily provable that the optimizations do not allow for unrealized functions.

Although it is unclear whether or not previous works used the same implementation and optimizations for $n < 5$, this implementation is the same, and uses the same optimizations for all values of n.

The number of classes for $n = 3,4$, as calculated by this implementation, equals the known number of classes calculated previously for the same values of n, but only differs once $n = 5$. This weighs favourably towards the method in which the functions are realized, as errors in the algorithm might also be apparent in the lower values of n.

In section 5.3.2, statistics on the number of functions per pre-filter group and class are examined, and the distribution of functions among the classes and groups appears to be reasonable. Section 5.3.1 indicates a 1-to-1 relationship between 191 of the 206 of the classes discovered in this research, and 191 of 191 classes indicated in [2]. A reasonable explanation is that the implementation in [2] over pruned the problem and combined a few classes that should have been separate.

The number of transformations, or rules, generated by this implementation has also been independently checked using a separate implementation, as seen in Section 5.3.3.

Although the number of operations, and not the operations themselves, have been checked, it is necessary for at least two operations to be incorrect for the list to be incorrect, yet poses the same number of operations. For the generated list to be incorrect for $n = 5$, it is likely that errors would also occur for $n < 5$.

Individually, these points do not prove the correctness of the data, but combined they make a very strong case for the results achieved in this research.

5.6 Summary

This research indicates that there are 15 new spectral classes, not previously identified in any work. There is strong evidence that these 15 previously unidentified classes had been inadvertently combined with other classes in previous work, but as the number of functions for values of n increases double-exponentially, the results cannot be checked with brute force.

The optimizations employed in this research significantly reduce the number of operations needed to classify all 2^{2^n} functions compared to the unoptimized problem.

Chapter 6 - Conclusion And Future Work

6.0 Introduction

As with all research, this work has introduced many questions that are beyond the scope of this thesis. Further research will be needed to prove these results, and to classify functions with $n > 5$ input variables. It is likely that this work will require new approaches, as current approaches tend not to scale well. Additional use of technology, such as specialized co-processors, or distributed computing, may also be required to perform all of the needed calculations.

6.1 Future Work

Although the analysis in Chapter 5 instils a high level of confidence in the results produced in this research, further work could be done to further improve the confidence in these results. Several approaches, which are beyond the scope of this research, could be used to further increase confidence, if necessary. The following section describes some of these approaches.

6.1.1 Implementation Analysis

An independent code review of the implementation in Appendix A could increase the confidence in the algorithm. Additionally, a code review could identify logic errors, if they exist.

Re-implementation of spectral classification in the spectral domain could also increase the confidence in the results. A separate implementation would be best served if it was independent of this implementation and did not share a code base.

6.1.2 Theoretical

As stated in section 5.3.3, the general equation for the number of Type 4 transformations is currently unknown. To formulate this, the number of linearly independent vector sets for all vector sets where the diagonal contain true bits (Figure 17) must be determined. This would likely be a large enough problem to constitute its own research thesis.

With current technology it is likely that the algorithm used in this research will not scale to classification for $n > 5$, and therefore a new approach is needed. Prediction based on lower values of n could provide a lower and/or upper bound for the number of classes for a given number of input variables. In the best case, prediction could provide an exact number of classes.

Current work to partition the problem for parallelization, pre-filtering, is described in section 4.2.1. The pre-filter approach does split the functions into groups, but these groups are not small enough to greatly reduce the running time when $n > 5$. Some method for increasing the granularity of the pre-filter could make a highly parallel implementation feasible.

6.1.3 Improvements To The Approach

The biggest enemy to this approach is primary memory usage. To reduce CPU and I/O overhead due to swapping, or other class storage schemes, values for all starting functions are placed in main memory. In other words, values for 2^{2^n-1} functions must be stored in main memory. For $n \leq 5$, this is manageable, but for $n > 5$ this will simply not be practical for the foreseeable future. To calculate $n > 5$, some other class storage scheme will be needed; likely one that increases processing time.

Allowing the classification and transformation to be run in parallel could reduce the overall running time of extra processing at the classification stage, but at the risk of duplicating processing on certain classes. In other words, if a starting function is realized in the previous pass, but not yet classified, realization of all functions from that starting function will occur, even though its class is already known.

Implementation of this approach for parallel processing to be run on distributed systems could increase the overall throughput of the application. Parallel processing is vary favourable for this kind of work because the realization of all functions for a given class and starting function does not depend on any external data until the end of the processing where it must be merged into the class list. Failing an improvement in granularity, a merging scheme is needed that allows multiple sets of realized functions, that potentially reside within the same class, to be classified. The problem with a merging scheme is that there is wasted work when there is a class collision (in both merging, and calculating it all in the first place since one pass is enough to realize all functions in that class).

Finally, implementation of an FPGA accelerated solution would likely yield improved real-life results over an entirely software solution assuming I/O bottlenecks can be minimized.

6.2 Conclusion

The research for this thesis continues where [2] left off with calculating the spectral classes for $n = 5$. The goals for this research comprises of:

4) Develop a new approach for computing spectral classes, and implement this approach.
5) Independently reproduce and verify the results published in [2], the spectral classes for $n = 5$, ensuring that it is a valid basis for future work. This goal is to be carried out using the results of goal 1.
6) Use the knowledge gained in goal 2 to investigate the possibility of computing the spectral classes for functions with values of n greater than 5. If it is feasible to compute the spectral classes for $n > 5$, then provide the classes for as many values of n as possible.

This research successfully achieves these goals, which are presented in detail in this thesis. A summary of the findings for each of these goals is as follows:

1) A new approach to computing spectral classes is proposed where the spectral operations and classification are performed entirely in the functional domain.
    a. This is significant as the classification with this approach completely avoids spectral transformations, except for the analysis and direct comparison between the results and previous work.
    b. The concept of Rules is introduced. A model created to represent the spectral operations in the functional domain
    c. An optimization approach is proposed where the data set (the starting functions) is reduced rather than pruning the algorithm, allowing for feasible running times for $n = 5$.

2) A discrepancy between the results produced in this thesis, and the list of classes published in [2] was found.

   a. A list of 15 new spectral classes with the same signatures as classes found in [2] is been tabulated. It appears that this is due to class splitting, much like the findings in [4].

3) With current technology, it is not possible to compute the spectral classes for $n > 5$ using currently known techniques. Future work on algorithms and approaches, and advances in technology are needed before this will become feasible.

Bibliography

[1] J. E. Rice. “Autocorrelation Coefficients in the Representation and Classification of Switching Functions.” Ph.D Thesis, University of Victoria, 2003.

[2] S. L. Hurst, D. M. Miller, and J. C. Muzio. “Spectral Techniques in Digital Logic.” Academic Press, Inc., Orlando, Florida, 1985.

[3] E.D Nering, “Linear Algebra and Matrix Theory.” John Wiley and Sons, Inc., London, 1965.

[4] S. L. Hurst. “The Logical Processing of Digital Systems.” Crane, Russak & Company, Inc., New York, 1978.

[5] K. G. Beauchamp. “Applications of Walsh and Related Functions.” Academic Press, London, 1984.

[6] “The On-Line Encyclopedia Of Integer Sequences.” 18 April 2007. <http://www.research.att.com/~njas/sequences/> (18 April 2007).

[7] C. Edwards. "The Application of the Rademacher-Walsh Transform to Boolean Function Classification and Threshold Logic Synthesis." in *IEEE Trans. on Comp*. pages 48-65, 1975.

[8] S. L. Hurst. “The Interrelationships Between Fault Signatures Based Upon Counting Techniques.” *Developments in Integrated Circuit Testing.* ed. D. M. Miller, Academic Press, London, 1987.

[9] D. M. Miller. “An Improved Method for Computing a Generalized Spectral Coefficient.” in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.* Volume 17, Number 3. pages 233-238, March 1998.

[10] M. A. Thornton, R. Drechsler, D. M. Miller, and W. Townsend. "Computing Walsh, Arithmetic and Reed-Muller Spectral Decision Diagrams Using Graph Transformations." GLVLSI 2002.

[11] R. Hyde. "Write Great Code Volume 2: Thinking Low-Level, Writing High-Level." No Starch Press, San Francisco, 2004.

Appendix A - Implementation

A.0 Introduction

This chapter is intended to provide an introduction to the implementation of the application used to produce the classification results for $n = 3,4,5$. Pseudocode is provided for the more significant sections of the implementation, along with explanations of why the decisions were made and why it works. This section should also help explain the full source code provided in Appendix C.

A.1 Language

The implementation for this application was accomplished in stages using a prototype approach, which was crucial to the choice of the programming language. Originally, the goal was to develop an approach to classify all 2^{2^n} functions for $n = 3$. As $n = 3$ only produces 256 functions, and implementation was not considered for $n > 3$, Java was chosen because of its ease of use and vast array of libraries. In this early stage, all of the processing was directed at the transformation and classification of the function, and the rules were computed by hand. In addition to hand-computed rules, the code was simple and many assumptions were hard coded.

Once successfully accomplishing spectral classification for $n = 3$, the application was modified to calculate the classes for $n = 4$. At first, this seemed like a trivial change, but it quickly became evident that creating the Type 4 rules for $n = 4$ by hand is not practical.

Code was written to generate all three rule types. Although some assumptions were made in order to save development time initially, scalability was a consideration in the design of this class.

The new rule generation code allowed all 2^{2^n} functions for $n = 4$ to be classified correctly. In order to calculate $n > 4$ it was determined that the code must be re-written using an object oriented approach and to be scalable rather than being re-written for each value of n. Using Java's dynamic structures such as ArrayLists (equivalent to the Vector data type in C++) and Maps, the entire application was re-written in a format structure similar to final structure described in section A.2, including removing the assumptions made by the existing Rule generation.

At this point, it was determined that Java was simply too slow and had too much overhead to finish the calculations in any reasonable amount of time and with the resources available. It was decided that the size of the problem for $n = 5$ was simply too large to be implemented in Java. As an alternative to Java, C++ was determined to be an appropriate replacement as it offered similar libraries and capabilities, but also offered decreased overhead, increased performance for certain types of calculations, and the ability to write at a lower level if needed.

The entire application was ported over to C++ using similar data types and structures as the original Java version. As a result, the C++ version of the code experienced similar performance issues as the Java version. Although the C++ version was able to complete more of the problem than the Java version in the same amount of time, the amount of time it would need to completely classify all 2^{2^5} functions was still not good enough. In fact, the application needed to be highly optimized, as described in section A.3, in order to achieve usable running times.

In the end, the C++ low overhead and ability to program at a low level allowed for a more efficient implementation of the algorithms. Additionally, C++'s vast libraries allowed for the use of built-in functions that decreased the amount of needed code, and therefore reduced the likelihood of errors.

A.2 Program Structure

The structure of this application is split into four major classes: Pre-filtering, Rule generation, Classification, and Transformation. In addition to the four class types, there is a main application file and a library of common tools used by all of the classes.

The implementation uses a modular approach and the flow of work can be visualized by the diagram in Figure 20.

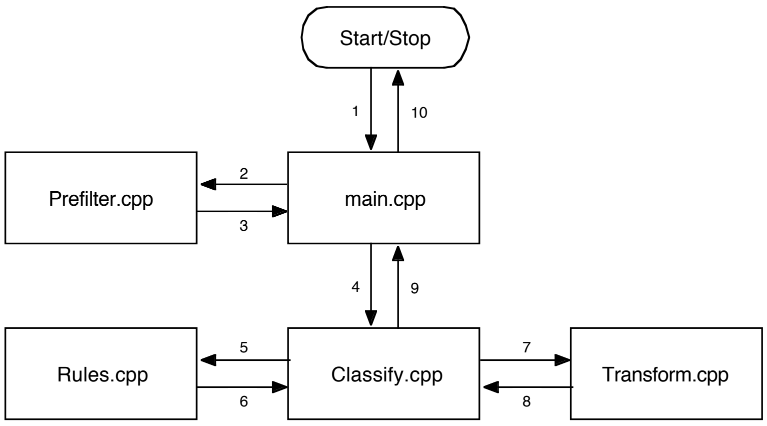

Figure 20 – Implementation flow

The first step in the classification process is for main.cpp to setup the environment with some simple calculations based on the number of input variables. Next, in step 2 of Figure 20, main.cpp calls Prefilter.cpp that is designed to separate all of the 2^{2^n} possible functions into groups based on the number of true bits within its binary representation.

Prefilter.cpp creates 2^{n-1} files on the secondary storage device and sends the list of files back to main.cpp in step 3. The main.cpp section then sends this list of files to Classify.cpp, as seen in step 4. The purpose of Classify.cpp is to do spectral classification for all of the functions in the pre-filtered function files. The first step that Classify.cpp must do is create a list of rules, or transformations, that must be applied to each function in the list, which is accomplished by a call to Rules.cpp in step 5. Rules.cpp compiles a list of transformations for Types 1, 2, and 4 (the Type 3 transformation simply inverts all of the values and therefore does not need a set of rules) that are returned to Classify.cpp. For each function in the pre-filter groups, Classify.cpp sends the function number and set of rules to Transform.cpp, as seen in step 7. Transform.cpp applies the rules to function and keeps a list of all functions that are generated throughout the transformation process. In step 8, this list of generated functions is returned to Classify.cpp to be incorporated into the list of classes. Steps 7 and 8 are done once for every single function in the current pre-filter group. After Classify.cpp has processed a pre-filter group, the functions are written to secondary storage with the corresponding class number that was assigned during the process. Once all of the pre-filter groups have been processed, control is passed back to main.cpp, the classes for 0 true bits are printed to secondary storage (this is a special case and no calculations are needed). At this point, main.cpp can clean up any temporary files, such as the pre-filter group files, and quit.

A.2.1 main.cpp

Main.cpp is the main module of the application and is responsible for setting up the environment, such as global variables, and pre-filter temporary file names. First, this module calculates the number of functions needed based on n. Second, the file names for the temporary files are defined, based on the number of functions in each pre-filter group. The prefilter module is then called, and this information is passed to the classification module. Finally, the main module does housekeeping and prints out Function 0.

A.2.2 Prefilter Class

Based on the theorem described in section 4.3.1, Type 1, Type 2 and Type 4 transformations cannot change the number of true bits (“ones”) in a function. Therefore, one can say that if all functions are separated into groups based on the number of true bits, a class cannot have functions that exist in more than one of these groups. The only other transformation is Type 3, and although it does change the number of true bits, the output is a simple inversion. The number of pre-filter groups can be reduced by half by grouping the inverted output with the main functions. For example, Functions a and b where $a = 1110000$ and $b = 11100011$ can be grouped together because a has 3 true bits, and the inversion of b has 3 true bits.

```
SET n to the number of input variables
CREATE prefilter_group_list with $2^{n-1}$ items

FOR each prefilter_group_list item
   CREATE group file with index number as file name

   FOR each function
      SET number_of_ones to 0

      FOR each bit of the function number
         IF the bit is true THEN
            INCREMENT number_of_ones
         ENDIF
      ENDFOR

      FOR each prefilter_group_list item index number
         IF number_of_ones equals the index number THEN
            WRITE function number to prefilter_group_file
         ELSEIF number_of_ones equals $2^n$-index number THEN
            WRITE function number to prefilter_group_file
         ENDIF
      ENDFOR

   ENDFOR

ENDFOR
```

Figure 21 – Pseudocode for pre-filter logic

In Figure 21, this calculation can be seen as the condition statement in the final FOR loop. As the number of true bits is a simple count, the inverse of the count is simply the number of true bits subtracted from 2^n.

A.2.3 Classify Class

The Classify class is responsible for reading each of the 2^{2^n} function numbers from the pre-filtered group files, assigning class numbers, and writing the classes to output files.

```
SET class_number_tracker to 0
SET increment_flag to false

CREATE classes_array with $2^{2^n}$ elements
CREATE final_class_number

FOR each pre-filter file
   CREATE temp_working array

   FOR each function in the pre-filter file
      INCREMENT class_number_tracker
      SET increment_flag to false

      IF this function has not already been found THEN
         CALL transformation class for this function
         SET temp_working array to results from transformation call
         SET increment_flag to true
         SET final_class_number to class_number_tracker

         FOR each item in temp_working array
            IF no value THEN
               CONTINUE
            ENDIF

            IF classes_array item has existing value THEN
               SET class_number to existing value
               SET increment_flag to false
               ENDFOR
            ENDIF

         ENDFOR

         FOR each item in temp_working array
            IF temp_working array item has value THEN
               SET classes_array item to final_class_number
            ENDIF
         ENDFOR

      ENDIF

      IF increment_flag is false THEN
         DECREMENT class_number_tracker
      ENDIF

   ENDFOR

   WRITE all found classes to output file

ENDFOR
```

Figure 22 – Pseudocode for classification logic

An array of 2^{2^n} elements is created to store the list of classes and initialized to an invalid class number. For each function number, the number is sent to the transformation class. The return value from the transformation class is a list of functions that can be achieved from this number by applying all of the Type 1-4 transformations. For each item in this list, we check the classes array to see if it has previously been assigned a class number. If any of the found functions exist in the classes array with a valid class number, all of new functions are marked in the classes array with the found class number. If none of the functions find a match in the classes array, the next sequential class number is assigned to all of the new functions.

A.2.4 Rules Class

The Rules class creates a list rules, or transformations, that are used by the transformation class. These rules describe to the transformation class how bits the output vectors (the function numbers) must be interchanged in order to create a new function number in the same class as the original. Rules for Types 1, 2, and 4 are created and passed back to the Classify class. There is no need to create a set of rules for Type 3 transformations as the output is simply inverted once.

A.2.4.1 Type 1: Permutation Of Input Variables

The permutation class makes use of the *next_permutation()* method built in to the C++ STL. This method returns every possible permutation of a supplied array. Based on these values, the new output array can be determined by a working copy of the function, the original starting function and the iteration of the loop.

```
CREATE original array with n elements
CREATE t1_rules stack

SET num to 0
FOR each element in original array
   SET original array element to num
   INCREMENT num
ENDFOR

REPEAT
   CREATE working_temp_rule stack

   SET outer_loop_value to 0

   FOR 0 through $2^n$
      CREATE orig_temp array of size n
      CREATE new_temp array of size n
      CALL itobv to convert outer_loop_value to array
      SET orig_term to value returned by itobv

      SET inner_loop_value to 0
      FOR 0 through n
         SET new_term with index of inner_loop_value to orig_temp_
                with index of original with index of inner_loop
         INCREMENT inner_loop_value
      ENDFOR

      CALL bvtoi to convert new_term back to a binary value
      SET val to value returned by bvtoi
      PUSH val to working_temp_rule
      INCREMENT outer_loop_value

   ENDFOR

   PUSH working_temp_rule to t1_rules

   CALL next_permutation function
UNTIL no next_permutation value
```

Figure 23 – Pseudocode for Type 1 rule generation

A.2.4.2 Type 2: Negation Of Input Variables

To create a list of 2^n Type 2 transformations, the value resulting from an XOR between the list index number and each of the truth table entries for the given number of variables, as seen in the example for $n = 3$ in Figure 24.

$$\begin{gathered}
011 \oplus 000 = 011 \rightarrow d \\
011 \oplus 001 = 010 \rightarrow c \\
011 \oplus 010 = 001 \rightarrow b \\
011 \oplus 011 = 000 \rightarrow a \\
011 \oplus 100 = 111 \rightarrow h \\
011 \oplus 101 = 110 \rightarrow g \\
011 \oplus 110 = 101 \rightarrow f \\
011 \oplus 111 = 100 \rightarrow e
\end{gathered}$$

Figure 24 – Example of array item 4 (011)

This XOR calculation is done for each of the 2^n rules, with the indices 0 through n (in its binary representation).

```
CREATE type_2_rules stack
SET outer_loop_value to 0

FOR 0 through 2^n
   CREATE working_temp_rule stack
   SET inner_loop_value to 0

   FOR 0 through 2^n
      SET val to inner_loop_value XOR outer_loop_value
      PUSH val to working_temp_rule
      INCREMENT inner_loop_value
   ENDFOR

   PUSH working_temp_rule to type_2_rules

   INCREMENT outer_loop_value
ENDFOR
```

Figure 25 – Pseudocode for Type 2 rule generation

A.2.4.3 Type 4: Variable Replacement With XOR

Conceptually, the Type 4 rule generation method creates a list of all possible functions that can be created with the given number of input variables. The method then traverses through the list, checking whether the combination is linearly independent. If a function is found to be linearly independent, and therefore a valid function, it is added to the list of known valid functions. Based on this list of valid functions, the resulting output vectors of these functions are added to the Type 4 rule list.

The implementation of the Type 4 rule generation is split into two main methods. First, a list of all possible input variable combinations must be created. Secondly, based on these input combinations, the validity of the function must be determined, and an output vector for the function must be created and added to Type 4 rule list.

A.2.4.3.1 Variable Input Combination List

The Type 4 transformation involves replacing an individual variable with itself XORed with one or more of the other input variables, as outlined in Section 2.3.2.2. To implement this so that all possible combinations are considered, it is easiest to create a table of all possible substitutions for a given input variable. Each row in Table 18 represents all possible substitutions for the variable listed in the first column. To consider all possible combinations of all possible input variables, a list must be compiled of all combinations from the table that contains one selection from each row.

x_3 (0001)	$x_3 \oplus x_2$ (0011)	$x_3 \oplus x_1$ (0101)	$x_3 \oplus x_2 \oplus x_1$ (0111)	$x_3 \oplus x_0$ (1001)	$x_3 \oplus x_2 \oplus x_0$ (1011)	$x_3 \oplus x_1 \oplus x_0$ (1101)	$x_3 \oplus x_2 \oplus x_1 \oplus x_0$ (1111)
x_2 (0010)	$x_3 \oplus x_2$ (0011)	$x_2 \oplus x_1$ (0110)	$x_3 \oplus x_2 \oplus x_1$ (0111)	$x_2 \oplus x_0$ (1010)	$x_3 \oplus x_2 \oplus x_0$ (1011)	$x_2 \oplus x_1 \oplus x_0$ (1110)	$x_3 \oplus x_2 \oplus x_1 \oplus x_0$ (1111)
x_1 (0100)	$x_3 \oplus x_1$ (0101)	$x_2 \oplus x_1$ (0110)	$x_3 \oplus x_2 \oplus x_1$ (0111)	$x_1 \oplus x_0$ (1100)	$x_3 \oplus x_1 \oplus x_0$ (1101)	$x_2 \oplus x_1 \oplus x_0$ (1110)	$x_3 \oplus x_2 \oplus x_1 \oplus x_0$ (1111)
x_0 (1000)	$x_3 \oplus x_0$ (1001)	$x_2 \oplus x_0$ (1010)	$x_3 \oplus x_2 \oplus x_0$ (1011)	$x_1 \oplus x_0$ (1100)	$x_3 \oplus x_1 \oplus x_0$ (1101)	$x_2 \oplus x_1 \oplus x_0$ (1110)	$x_3 \oplus x_2 \oplus x_1 \oplus x_0$ (1111)

Table 18 – Logical representation of lookup table for $n = 4$

The logical representation, as shown in Table 18, can be converted into a binary representation by creating a binary number of length n, and using each bit to represent a different input variable. In our representation, the least significant bit represents the first variable. For example, $x_3 \oplus x_2 \oplus x_1$ would be equivalent to `0111`.

	2^{n-1}								
x_3	1	3	5	7	9	11	13	15	
x_2	2	3	6	7	10	11	14	15	n
x_1	4	5	6	7	12	13	14	15	
x_0	8	9	10	11	12	13	14	15	

Table 19 – Decimal representation of lookup table for $n = 4$

The binary representation shown in the parentheses in Table 18 can instead be represented by the decimal integer which is beneficial in implementation. This table can be easily generated programmatically.

x_0			x_1			x_2			x_3			x_4	
1	00001	$+2^0$	2	00010	$+2^1$	4	00100	$+2^2$	8	01000	$+2^3$	16	10000
3	00011	$\rightarrow$	3	00011	$+2^1$	5	00101	$+2^2$	9	01001	$+2^3$	17	10001
5	00101	$+2^0$	6	00110	$\rightarrow$	6	00110	$+2^2$	10	01010	$+2^3$	18	10010
7	00111	$\rightarrow$	7	00111	$\rightarrow$	7	00111	$+2^2$	11	01011	$+2^3$	19	10011
9	01001	$+2^0$	10	01010	$+2^1$	12	01100	$\rightarrow$	12	01100	$+2^3$	20	10100
11	01011	$\rightarrow$	11	01011	$+2^1$	13	01101	$\rightarrow$	13	01101	$+2^3$	21	10101
13	01101	$+2^0$	14	01110	$\rightarrow$	14	01110	$\rightarrow$	14	01110	$+2^3$	22	10110
15	01111	$\rightarrow$	15	01111	$\rightarrow$	15	01111	$\rightarrow$	15	01111	$+2^3$	23	10111
17	10001	$+2^0$	18	10010	$+2^1$	20	10100	$+2^2$	24	11000	$\rightarrow$	24	11000
19	10011	$\rightarrow$	19	10011	$+2^1$	21	10101	$+2^2$	25	11001	$\rightarrow$	25	11001
21	10101	$+2^0$	22	10110	$\rightarrow$	22	10110	$+2^2$	26	11010	$\rightarrow$	26	11010
23	10111	$\rightarrow$	23	10111	$\rightarrow$	23	10111	$+2^2$	27	11011	$\rightarrow$	27	11011
25	11001	$+2^0$	26	11010	$+2^1$	28	11100	$\rightarrow$	28	11100	$\rightarrow$	28	11100
27	11011	$\rightarrow$	27	11011	$+2^1$	29	11101	$\rightarrow$	29	11101	$\rightarrow$	29	11101
29	11101	$+2^0$	30	11110	$\rightarrow$	30	11110	$\rightarrow$	30	11110	$\rightarrow$	30	11110
31	11111	$\rightarrow$	31	11111	$\rightarrow$	31	11111	$\rightarrow$	31	11111	$\rightarrow$	31	11111

Table 20 – Lookup table generation for $n = 5$

As seen in Table 20, the lookup table, like the one in Table 19, can be generated by first filling the first row with the odd integers from 0 to 2^n. The remaining $n-1$ rows are filled in one at a time, using the previous rows as a reference. Using a "skip" value of 2^k where k goes from 0 to $n-2$, the table can be programmatically generated by adding the skip value to the value of the item in the same column, but the row above. This occurs the same number of times as the value of the skip value, and then the same number of columns are skipped. This process continues until all column elements are filled. For each successive row, k of the skip value is increased. Table 20 illustrates this pattern graphically. The pseudocode in Figure 26 demonstrates how this lookup table can be created at runtime for any value of n, rather than hard coding it for each value considered.

```
CREATE arr 2D array of width 2^(n-1) and height n

INITIALIZE arr values to 0

SET skip_value to 0

SET loop_value to 0
FOR 0 through width of arr
   SET arr value of index width of 0 and index height of loop_value to_
          loop_value + 1 + skip_value
   INCREMENT skip_value
   INCREMENT loop_value
ENDFOR

SET loop_value to 1
FOR 1 through height of arr
   SET skip to 2^(loop value-1)
   SET inner_loop_value to 0
   FOR for 0 through width of arr
      FOR 0 through skip_value
         SET arr value of index width loop_value and height of_
                inner_loop_value to value of arr with index width_
                   of (loop_value - 1) and height of inner_loop_
                      + skip_value
         INCREMENT inner_loop_value
      ENDFOR

      FOR 0 through skip_value
         SET arr value of index width loop_value and height of_
                inner_loop_value to value of arr with index width_
                   of (loop_value - 1) and height of inner_loop_
         INCREMENT inner_loop_value
      ENDFOR
   ENDFOR
   INCREMENT loop_value
ENDFOR
```

Figure 26 – Pseudocode for lookup table creation

Once the lookup table has been generated, a list of all possible combinations, using one item from each row, must be compiled.

```
CREATE type4_rules stack
CREATE temp array with 3 elements

CREATE check array with $2^3-1$ elements

SET check element 1 to the result of:
       (0 * temp index 1) OR (0 * temp index 2) OR (1 * temp index 3)

SET check element 2 to the result of:
       (0 * temp index 1) OR (1 * temp index 2) OR (0 * temp index 3)

SET check element 3 to the result of:
       (0 * temp index 1) OR (1 * temp index 2) OR (1 * temp index 3)

SET check element 4 to the result of:
       (1 * temp index 1) OR (0 * temp index 2) OR (0 * temp index 3)

SET check element 5 to the result of:
       (1 * temp index 1) OR (0 * temp index 2) OR (1 * temp index 3)

SET check element 6 to the result of:
       (1 * temp index 1) OR (1 * temp index 2) OR (0 * temp index 3)

SET check element 7 to the result of:
       (1 * temp index 1) OR (1 * temp index 2) OR (1 * temp index 3)

IF any element in check is not equal to 0 THEN
   PUSH temp to type4_rules
ENDIF
```

Figure 27 – Pseudocode for linear independence check for $n = 3$

The basic logic for the linear independence check of an input variable combination is the literal implementation of equation (2.18), as seen in Figure 27.

```cpp
// First run
for (int m = 1; m < vecLen; m++) {
    a[m] = tmp[0] * ((m >> (numRow-1)) & 1);
}

// For all the consecutive calculations
for (int m = 1; m < vecLen; m++) {
    for (int k = 1; k < numRow; k++) {
        a[m] ^= tmp[k] * ((m >> (numRow-k-1)) & 1);
    }
}
```

Figure 28 – C++ code to check linear independence for any value of n

This specific example can be implemented to encompass any value of n, as seen in Figure 28. Although the nested loops and extensive use of bitwise operators obscure the intention of this code, it is still essentially a literal implementation of equation (2.18).

A.2.3.4.2 Type 4 Rule Generation

This method converts the valid XORed variable combinations provided by the *type4List()* method from a binary representation into rule format as expected by the transformation class. The core of this method is the line of code displayed in Figure 29, which is taken from the source code in section Appendix C.3.2.

```
tmp[m] = tmp[m] ^ (((t4List[i][m] >> j) & 1) * ((k >> j) & 1));
```

Figure 29 – Innermost for loop logic

Although the bitwise operations and nested loops obscure this code, it simply isolates the individual bits of each column item in the t4List array for a given row, and multiplying that bit by the isolated bit for that iteration of the loop.

```
tmp[0] = tmp[0] ^ (((t4List[i][0] >> 0) & 1) * ((0 >> 0) & 1));
tmp[0] = tmp[0] ^ (((t4List[i][0] >> 0) & 1) * ((1 >> 0) & 1));
tmp[0] = tmp[0] ^ (((t4List[i][0] >> 0) & 1) * ((2 >> 0) & 1));
tmp[0] = tmp[0] ^ (((t4List[i][0] >> 0) & 1) * ((3 >> 0) & 1));

tmp[0] = tmp[0] ^ (((t4List[i][0] >> 1) & 1) * ((0 >> 1) & 1));
tmp[0] = tmp[0] ^ (((t4List[i][0] >> 1) & 1) * ((1 >> 1) & 1));
tmp[0] = tmp[0] ^ (((t4List[i][0] >> 1) & 1) * ((2 >> 1) & 1));
tmp[0] = tmp[0] ^ (((t4List[i][0] >> 1) & 1) * ((3 >> 1) & 1));

tmp[1] = tmp[1] ^ (((t4List[i][1] >> 0) & 1) * ((0 >> 0) & 1));
tmp[1] = tmp[1] ^ (((t4List[i][1] >> 0) & 1) * ((1 >> 0) & 1));
tmp[1] = tmp[1] ^ (((t4List[i][1] >> 0) & 1) * ((2 >> 0) & 1));
tmp[1] = tmp[1] ^ (((t4List[i][1] >> 0) & 1) * ((3 >> 0) & 1));

tmp[1] = tmp[1] ^ (((t4List[i][1] >> 1) & 1) * ((0 >> 1) & 1));
tmp[1] = tmp[1] ^ (((t4List[i][1] >> 1) & 1) * ((1 >> 1) & 1));
tmp[1] = tmp[1] ^ (((t4List[i][1] >> 1) & 1) * ((2 >> 1) & 1));
tmp[1] = tmp[1] ^ (((t4List[i][1] >> 1) & 1) * ((3 >> 1) & 1));
```

Figure 30 – Innermost loop logic for $n = 2$ (unfolded loops)

By undoing some of the nested loops for an example of $n = 2$, as in Figure 30, the code becomes clearer. For each row in the t4List array, 16 assignment operations take place on tmp (8 to each element of the array). Each line simply takes the current value of tmp, does a bitwise OR with the result of the isolated variable multiplied by the equivalent bit within the integers 0 through 2^n.

```
CREATE t4_rules stack
CREATE initial_order stack

SET loop_count to 0

FOR 0 through $2^n$
   PUSH loop_count to initial_order
   INCREMENT loop_count
ENDFOR

PUSH initial_order to t4_rules

CALL genTypeFourList
SET t4_list to returned 2D (width of n) array from genTypeFourList

SET loop_i to 0
FOR each item in t4_list
   CREATE temp_working_rule stack
   SET loop_k to 0

   FOR 0 through $2^n$
      CREATE temp stack
      SET loop_m to 0
      FOR 0 through n
         PUSH 0 to temp
         SET loop_j to 0
         FOR 0 through n
            SET shift_item to t4_list with index loop_i and loop_m SHIFT_
                   right by loop_j all AND by 1
            SET or_item to shift_item multiplied by loop_k SHIFT right_
                   by loop_j all AND by 1
            SET temp with index of loop_m to the value in temp with index_
                   of loop_m OR by or_item
            INCREMENT loop_j
         ENDFOR
         INCREMENT loop_m
      ENDFOR
      CALL bvtoi to convert temp to binary format
      SET t_result to bvtoi returned value
      PUSH to temp_working_rule
      INCREMENT loop_k
   ENDFOR
   PUSH temp_working_rule to t4_rules
   INCREMENT loop_i
ENDFOR
```

Figure 31 – Pseudocode for rule generation

The pseudocode provided in Figure 31 places the code in Figure 29 in context of the nested for loops.

A.2.5 Transform Class

The Transform class is responsible for producing a list of all possible functions that can be generated when all four transformation types are applied to a single starting function. The main transformation method may be called up to 2^{2^n} times by the classification class: once for each of the 2^{2^n} starting functions.

```
METHOD transform

   CREATE classes_array with $2^{2^n}$ elements
   CALL type_1 method with function_number
   RETURN classes_array to caller

ENDMETHOD

METHOD type_1
   FOR each item in type1_rules
      CALL swapBits method with function_number and rule
      SET num to value returned by swapBits
      CALL type_2 method with num
   ENDFOR
ENDMETHOD

METHOD type_2
   FOR each item in type2_rules
      CALL swapBits method with function_number and rule
      SET num to value returned by swapBits
      CALL type_3 method with num
   ENDFOR
ENDMETHOD

METHOD type_3

      SET mask to $2^n$ true bits
      CALL type_3 method with function_number
      SET num to function_number XOR mask
      CALL type_3 method with num
ENDMETHOD

METHOD type_4
   FOR each item in type4_rules
      CALL swapBits method with function_number and rule

      IF value returned by swapBits is smaller than $2^{2^n}$ THEN
         SET class_array element with index of function_number to true
      ENDIF
ENDMETHOD

METHOD swapBits

   CREATE original array with $2^n$ bits
   FOR all bits of function_number
      SET original element to value of that bit
   ENDFOR

   SET new_function_number to 0;

   FOR all elements of order array
      SET new_function_number to new_function_number shifted left by 1_
             place and OR with value of original with the index of the_
                rule order with the index of the current element
   ENDFOR
ENDMETHOD
```

Figure 32 – Pseudocode for transformation logic

For each function, the transformation begins by calling the Type 1 classification method. The Type 1 method traverses this list of rules, applying them to the function and storing the new functions in a working list. This working list is globally available to all of the transformation methods the transformation class. For each item in a rule list, the current transformation method calls the method below it. In other words, if the rules of all types were to be stored in a tree, where each level represented a transformation type, as in Figure 18, the order in which the rule would be processed would be the same as if one were to perform a pre-order traversal. Once the final rule is processed, the list of functions accumulated throughout the traversal is passed back to the classification class.

A.3 Optimizations

A.3.1 Reducing The Problem

As previously discussed, Type 3 transformations simply invert the values in the output vector, as seen in Figure 33. On every iteration of the classification process, all four transformation types are applied to each function. Since Type 3 is applied to every function, by the time half of the functions have been processed, all possible functions have been discovered and classified.

$$\begin{array}{c} 00 \leftrightarrow 11 \\ 01 \leftrightarrow 10 \\ \hline 10 \leftrightarrow 01 \\ 11 \leftrightarrow 00 \end{array}$$

Figure 33 – All possible functions for $n = 1$

If all functions have been discovered by the time 2^{2^n-1} functions have been processed, processing the remaining 2^{2^n-1} is redundant. Based on this observation, the implementation only considers the first 2^{2^n-1} functions. Without considering any other optimizations, this could reduce the running time to half of the original.

Using this same observation, it becomes obvious that storing values above 2^{2^n-1} is also redundant. Reducing memory requirements to half at $n = 5$ is significant as it allowed us to fit the application into the primary storage; something that had not been possible with the available equipment. The ability to store all working data in primary storage significantly reduces the overhead that would be needed for a system that relies on caching to a secondary storage device.

The optimization, used to reduce the amount of calculations, is to check if the current starting function has previously been discovered. If this function has previously been discovered, applying the transformations will simply return other functions that have already been discovered. As this is redundant processing, the expensive transformation processing can be eliminated.

A.3.2 Programming Techniques

This application needed to be highly optimized in order to run on the available resources, and also complete in a reasonable amount of time. The following sections describe techniques used to optimized portions of code identified to be bottlenecks when using runtime profilers.

A.3.2.1 Dynamic Vs. Fixed Data Structures

As discussed in Appendix A.1, a significant portion of the implementation was focused around reducing the overhead associated with programming language. The original C++ version of this application made extensive use of the C++ STL's dynamic data structures. The advantage of these structures is that they provide the developer easy to use tools, while also reducing the likelihood of errors. On the other hand, the disadvantages of these structures are based on their ease of use. For example, a STL Vector data structure can be used with any data type, and contains bounds checking to avoid errors due to incorrect access. These extra checks and layers of abstraction add a small amount of overhead to every call to that structure. For an average application, the number of calls to these structures are not that great, and the overhead does adversely affect the application. The benefits to the developer greatly outweigh the small change in performance. Unfortunately for certain applications, such as this implementation, the number of calls to these structures are in the billions or trillions rather than hundreds of thousands. In the case of this application for $n = 5$, the temporary working array in the Transform class is modified approximately 2.8 Billion times for each starting function. If no optimizations are applied to reduce the dataset, the application could modify this array 1.2×10^{19} times when considering all 2^{2^5} functions, and this only considers one data structure in one class. The overhead of a STL Vector, no matter how slight, adds up to be very significant over the entire running time.

As the dynamic data structures were not suitable for this application, fixed size data structures, such as the traditional array, had to be examined. The advantage of an array is that there is no level of abstraction that causes overhead for accessing or modifying the data contents. The biggest disadvantage of using an array is that the sizes must be pre-determined, or hard coded, which eliminates the scalability of the code. Additionally, the program stack is not large enough to contain an array large enough to store 2^{2^5}, or even 2^{2^5-1} functions. Fortunately, unlike Java, C++ allows direct allocation and manipulation of heap memory using the *new* operator (or the C-style *malloc* operator). Additionally, the new operator can allocate memory at runtime, so the scalability of the code is not lost. This memory can still be thought of, and accessed, like a regular array using the square brackets (Example: a[4]). Since all reads and writes to this dynamically created array are directly to memory, and not through accessor methods, there is no additional overhead.

The dynamically allocated arrays were used any time it would be accessed many times, size needed to be determined at run time, or items were simply too large to fit in the stack. There are cases where Vectors and regular arrays are used, but only scalability and performance were not adversely affected.

A.3.2.2 Bitwise Operators

Modern microprocessors rely on a certain small set of primitive operations to accomplish all of the calculations. Typically, a high level language is used and a compiler reduces these commands to combinations of the primitive operations. If an application is written considering "low-level execution of [the] high-level program," it is possible to create code that will run faster, or more memory efficient than what the compiler can derive from generic high-level code [11]. Modern compilers are well written and highly optimized, so it's not likely that for a random command, a person could write better code, but if an approach is taken to take advantage of these low level structures, it is possible to come up with a faster, or more memory efficient solution. For example, the output vector of a function's truth table consists of 2^n true or false values. One approach would be to create an array of 2^n Boolean values, and store each of the results in one of the elements. In C++, the Boolean data type is an 8-bit value. For $n = 4$, this array would take 128 bits of memory to store a single function. If one considers bitwise operators, this same function can be represented using a short integer, which takes only 16 bits. The result is that to simply store these functions, 8 times less memory is needed, which become significant when considering the size of the problem.

The previous example illustrated how being conscious of the low level operation of the processor allowed for more efficient memory usage, a second example can illustrate how bitwise operators can result in faster calculations. The individual bits can be easily accessed and using combinations of shift (>> or <<), AND (&), OR (|), and XOR (^)

operations. Consider the case where we want to check if any of the bits in the output vector are set to true. In Figure 34, a traditional approach using an array to store the output vector is used. In order to do this, each bit of the output vector would have to be checked.

```
int fn[4] = {0, 0, 1, 1}
int result = 0;
for (int i = 0; i < 4; i++) {
   if (fn[i] = 1) {
      result = 1;
   }
}

if (result) {
   // There are bits set to true
} else {
   // There are no bits set to true
}
```

Figure 34 – Check if any bits are set to true

If a bit was found to be true, we would set a flag, and then after checking each bit, we would test to see if the flag had been set. If we consider the same problem using bitwise operators, like in Figure 35, we can do an XOR between a mask of all true bits, and the binary representation of the function. If the integer happens to be anything other than 0, we know that one of the bits is true.

```
int fn = 3;    // Binary: 0011
int mask = 15; // Binary: 1111
if (fn ^ mask) {
   // There are bits set to true
} else {
   // There are no bits set to true
}
```

Figure 35 – Check if any bits are true using bitwise operators

Neither of these examples would have been achieved through compiler optimization since the problem is conceptually different. If one thinks in terms of a lower level of operation, bitwise operators can be a powerful tool in code optimization.

A.3.2.3 Picking Data Types That Fit

In order to calculate for $n = 5$, it was important to use the smallest data types possible to be able to squeeze the application into the available memory. One way to accomplish this is by the method mentioned in A.3.2.2, which represents functions as `ints` and `longs` rather than storing the values in an array or similar structure. Another example is the temporary list of classes maintained by the Transform class. Rather than storing an array of 2^{2^n-1} `ints`, which are 32-bits in length, an array of 2^{2^n-1} `bool` were used instead, which only take 8 bits of memory each.

A.3.2.4 Memory Vs. Clock Cycles

In software development, we must often balance how we use resources. On one hand, we can have a very fast algorithm, but it uses a lot of memory. On the other hand, we can have an algorithm that computes the same thing, but is very memory efficient. This memory efficiency usually comes at the cost of running speed.

This application at $n > 4$ is both CPU and memory intensive, which makes it very difficult to choose which side to sacrifice. Based on the resources available, it was determined that with the reduced problem set, it is possible to keep large arrays with values for each function in memory in the methods that are used frequently. This allows for the minimum amount of CPU overhead in sections of the code that are being run the most.

A.3.2.5 Object-Oriented Programming

Object-oriented programming is very good for creating scalable, modular, and reusable code. In the case of this implementation, if an improved algorithm for swapping the output vector bits is created, it is very simple to change without affecting the rest of the system. Unfortunately the overhead associated with this level of abstraction, much like the abstract data types in the STL that were previously discussed, can have a detrimental effect on the performance of the application.

To counter the effects of abstraction overhead, it was necessary to break some of the object-oriented conventions. Many of these choices included setting arrays and variables as globals rather than passing them as parameters. This allows the structures to be computed and allocated once, rather than when needed.

It was also necessary to pay careful attention to how data is passed, when needed. If a structure, such as a Vector, is passed by value rather than by reference, the overhead of the copy on each call of the method can add up very quickly. Every method in the commonly called methods (especially in the Transform class) were carefully examined to ensure all calls were pass-by-reference, and the effects were confirmed using a profiler.

A.3.2.6 Dividing The Problem

The pre-filtering that takes place in the application splits starting functions into self-contained groups. It has previously been shown that functions within these groups cannot be transformed into functions that are part of the other groups. This allows the groups to be classified separately.

Because of this property, it is possible that the groups could be processed in parallel, whether it is on multiple machines, or multiple threads. Although the pre-filtering has been implemented, multi-threading has not since for each thread, it would require an additional copy of the classes and temporary working arrays to be stored in memory, which would not have fit in the resources available. It is also possible to run this application on two separate machines, but we did not have two machines with sufficient primary storage for this application.

A.4 Problems

There are several issues that become apparent when trying to implement spectral classification of functions, as this is a huge problem, and one that grows double exponentially.

Due to this size, many compromises must be made during implementation. Critical decisions were made when choosing an appropriate programming language with sufficient abilities yet low overall overhead. Many times implementation required that good coding practices be ignored in order to achieve acceptable performance results.

Although it is possible to make some headway with programming techniques, the problem is still very large, and the problem size must be reduced. Some techniques involve making assumptions based on the overall properties of the problem, while other approaches involve dividing the problem into smaller portions. Unfortunately, the pre-filtering technique, although simple and relatively low overhead, is not scalable. As n increases to values over 5, these smaller portions are still far too large to be usable.

Scalability is not realistic with this approach as it relies heavily on keeping the entire problem in primary memory to reduce performance hits due to system overhead. The size of the problem becomes so great, that future work will likely have to rely on other approaches such as prediction.

A.5 Summary

The implementation used for this research calculates the spectral classes for $n = 3,4,5$ without needing any changes in optimizations or algorithms, therefore we are assured that for all considered values of n, the same approach is used.

The approaches used for this implementation mirror the concepts described in this thesis. There are a few specific cases where the implementation uses a slightly different approach, such as rolling two steps into a single step, in order to decrease memory usage, and running time; conceptually, though, the approaches are the same.

Spectral classification of Boolean functions is a difficult problem to implement due to the double exponential growth of the problem. Implementations that work well for $n \leq 5$ are not necessarily appropriate for $n > 5$. Optimizations that make $n = 5$ feasible do not necessarily help for larger values of n; in fact, the approaches used in this implementation to make $n = 5$ run fast enough to be feasible would make it impossible to calculate $n \leq 6$ with current technology.

The implementation was originally intended to be completely object oriented, but due to the size of the problem, optimizations that break standard object oriented approaches are needed. Although this implementation is not entirely object oriented, the modular intent of the implementation is maintained. This implementation could scale higher, given enough resources, but without some technological breakthroughs in hardware, it is not feasible.

Future work on implementation will likely require new approaches storing, indexing, and processing the data in order to make $n > 5$ possible, as well as further optimizations to the algorithm and approach.

Appendix B - Classes

B.0 Introduction

The results of this research have yielded different results than previous work in [2]. Much of the work in this research involves comparing the work in [2] to the current results to identify where the discrepancies occur. As the class list in [2] is not complete and only lists the spectral signature, rather than the entire spectrum of the canonical function, some reconstruction of working data from archives is needed. The result of this work is a complete list of spectral classes produced by this research, and a reconstructed list of the data produced in [2].

B.1 Complete Spectral Class List For $n = 5$

To make this spectral class list as comparable as possible to the list in [2], the same presentation format of the first order spectral coefficients, and a summary of the complete spectrum are used. Additionally, the decimal representation of the function, which is listed in the "Function Number" column of Table 21, has been added in order to identify the exact canonical function used for this classification. This function number can also be used to calculate the entire spectrum of the function if desired. In addition to the function number, a class number, as described in section 4.2.3, and a pre-filter group number have been added.

Finally, the equivalent canonical function number from [2] has been added under the heading of "Book," which uses a different approach to ordering and assigning canonical numbers. As there are more classes listed in this list than there are in [2], a dash (-) has been placed in the column rather than a number for certain functions.

Class Number	Group Number††	Function Number	s_0	s_1	s_2	s_3	s_4	s_5	Summary Of Complete Spectrum					Book‡‡
0	0	0	32	0	0	0	0	0	1 x32	31 x0				2
1	1	1	30	2	2	2	2	2	1 x30	31 x2				4
2	2	3	28	0	4	4	4	4	1 x28	15 x4	16 x0			7
3	3	7	26	2	2	6	6	6	1 x26	7 x6	24 x2			10
4	4	15	24	0	0	8	8	8	1 x24	7 x8	24 x0			17
5	4	23	24	4	4	4	8	8	1 x24	3 x8	16 x4	12 x0		14
6	5	31	22	2	2	6	10	10	1 x22	3 x10	4 x6	24 x2		21
7	5	279	22	6	6	6	6	10	1 x22	1 x10	10 x6	20 x2		25
8	6	63	20	0	4	4	12	12	1 x20	3 x12	12 x4	16 x0		29
9	6	287	20	4	4	8	8	12	1 x20	1 x12	4 x8	14 x4	12 x0	35
10	6	854	20	4	4	4	4	12	1 x20	1 x12	30 x4			87
11	6	65815	20	8	8	8	8	8	1 x20	6 x8	15 x4	10 x0		39
12	7	127	18	2	2	2	14	14	1 x18	3 x14	28 x2			42
13	7	319	18	2	6	6	10	14	1 x18	1 x14	2 x10	6 x6	22 x2	48
14	7	855	18	6	6	6	6	14	1 x18	1 x14	12 x6	18 x2		91
15	7	65823	18	6	6	10	10	10	1 x18	3 x10	9 x6	19 x2		54
16	7	66390	18	6	6	6	6	10	1 x18	1 x10	15 x6	15 x2		95
17	8	255	16	0	0	0	16	16	4 x16	28 x0				56
18	8	383	16	4	4	4	12	16	2 x16	2 x12	14 x4	14 x0		60
19	8	831	16	0	8	8	8	16	2 x16	8 x8	22 x0			64
20	8	863	16	4	4	8	8	16	2 x16	4 x8	16 x4	10 x0		100
21	8	65855	16	4	8	8	12	12	1 x16	2 x12	4 x8	14 x4	11 x0	72
22	8	66391	16	8	8	8	8	12	1 x16	1 x12	6 x8	15 x4	9 x0	105
23	8	197461	16	8	8	8	8	8	1 x16	12 x8	19 x0			108
24	8	197462	16	4	8	8	8	8	1 x16	8 x8	16 x4	7 x0		112
25	9	511	14	2	2	2	14	18	1 x18	3 x14	28 x2			41
26	9	895	14	2	6	6	10	18	1 x18	1 x14	2 x10	6 x6	22 x2	47
27	9	1911	14	6	6	6	6	18	1 x18	1 x14	12 x6	18 x2		90
28	9	65919	14	6	6	6	14	14	3 x14	1 x10	7 x6	21 x2		76
29	9	66367	14	2	10	10	10	14	2 x14	4 x10	4 x6	22 x2		81
30	9	66399	14	6	6	10	10	14	2 x14	2 x10	10 x6	18 x2		118
31	9	197463	14	6	10	10	10	10	1 x14	5 x10	7 x6	19 x2		124
32	9	197991	14	6	6	10	10	10	1 x14	3 x10	13 x6	15 x2		130
33	9	202070	14	10	6	6	6	10	1 x14	3 x10	13 x6	15 x2		134
34	10	1023	12	0	4	4	12	20	1 x20	3 x12	12 x4	16 x0		28
35	10	1919	12	4	4	8	8	20	1 x20	1 x12	4 x8	14 x4	12 x0	34
36	10	6014	12	4	4	4	4	20	1 x20	1 x12	30 x4			86
37	10	66047	12	4	4	4	16	16	2 x16	2 x12	14 x4	14 x0		59
38	10	66431	12	4	8	8	12	16	1 x16	2 x12	4 x8	14 x4	11 x0	71
39	10	67447	12	8	8	8	8	16	1 x16	1 x12	6 x8	15 x4	9 x0	104
40	10	197439	12	0	12	12	12	12	6 x12	10 x4	16 x0			84
41	10	197471	12	4	8	12	12	12	4 x12	4 x8	12 x4	12 x0		140
42	10	197999	12	4	4	12	12	12	4 x12	28 x4				143
43	10	198007	12	8	8	8	12	12	3 x12	6 x8	13 x4	10 x0		152
44	10	202071	12	12	8	8	8	12	4 x12	4 x8	12 x4	12 x0		147
45	10	202075	12	8	8	8	8	12	2 x12	8 x8	14 x4	8 x0		158
46	10	218454	12	12	4	4	4	12	4 x12	28 x4				154
47	10	218458	12	8	4	4	4	12	2 x12	8 x8	14 x4	8 x0		163
48	10	463702	12	8	8	8	8	8	1 x12	10 x8	15 x4	6 x0		167

†† Pre-filter group

‡‡ The equivalent class number in [2]

Class Number	Group Number	Function Number	s_0	s_1	s_2	s_3	s_4	s_5	Summary Of Complete Spectrum					Book
49	11	2047	10	2	2	6	10	22	1 x22	3 x10	4 x6	24 x2		20
50	11	6015	10	6	6	6	6	22	1 x22	1 x10	10 x6	20 x2		24
51	11	66559	10	2	6	6	14	18	1 x18	1 x14	2 x10	6 x6	22 x2	46
52	11	67455	10	6	6	10	10	18	1 x18	3 x10	9 x6	19 x2		53
53	11	71550	10	6	6	6	6	18	1 x18	1 x10	15 x6	15 x2		94
54	11	197503	10	2	10	10	14	14	2 x14	4 x10	4 x6	22 x2		80
55	11	198015	10	6	6	10	14	14	2 x14	2 x10	10 x6	18 x2		117
56	11	198519	10	6	10	10	10	14	1 x14	5 x10	7 x6	19 x2		123
57	11	202079	10	10	6	10	10	14	1 x14	5 x10	7 x6	19 x2		-
58	11	202095	10	6	6	10	10	14	1 x14	3 x10	13 x6	15 x2		129
59	11	218455	10	14	6	6	6	14	3 x14	1 x10	7 x6	21 x2		75
60	11	218459	10	10	6	6	6	14	1 x14	3 x10	13 x6	15 x2		133
61	11	463677	10	6	10	10	10	10	6 x10	10 x6	16 x2			170
62	11	463703	10	10	10	10	10	10	6 x10	10 x6	16 x2			175
63	11	471868	10	2	10	6	6	10	6 x10	10 x6	16 x2			178
64	11	471894	10	6	10	6	6	10	4 x10	16 x6	12 x2			182
65	12	4095	8	0	0	8	8	24	1 x24	7 x8	24 x0			16
66	12	6143	8	4	4	4	8	24	1 x24	3 x8	16 x4	12 x0		13
67	12	67583	8	4	4	8	12	20	1 x20	1 x12	4 x8	14 x4	12 x0	33
68	12	71551	8	8	8	8	8	20	1 x20	6 x8	15 x4	10 x0		38
69	12	197631	8	0	8	8	16	16	2 x16	8 x8	22 x0			63
70	12	198143	8	4	4	8	16	16	2 x16	4 x8	16 x4	10 x0		99
71	12	198527	8	4	8	12	12	16	1 x16	2 x12	4 x8	14 x4	11 x0	69
72	12	202111	8	8	8	8	12	16	1 x16	1 x12	6 x8	15 x4	9 x0	103
73	12	202621	8	8	8	8	8	16	1 x16	12 x8	19 x0			107
74	12	202622	8	4	8	8	8	16	1 x16	8 x8	16 x4	7 x0		111
75	12	218463	8	12	4	8	8	16	1 x16	2 x12	4 x8	14 x4	11 x0	70
76	12	218479	8	8	4	8	8	16	1 x16	8 x8	16 x4	7 x0		-
77	12	460663	8	8	8	12	12	12	4 x12	4 x8	12 x4	12 x0		139
78	12	463679	8	4	12	12	12	12	4 x12	4 x8	12 x4	12 x0		146
79	12	463711	8	8	8	12	12	12	3 x12	6 x8	13 x4	10 x0		151
80	12	463741	8	8	8	8	12	12	2 x12	8 x8	14 x4	8 x0		157
81	12	471869	8	4	12	8	8	12	2 x12	8 x8	14 x4	8 x0		162
82	12	471895	8	8	12	8	8	12	2 x12	8 x8	14 x4	8 x0		-
83	12	472423	8	8	8	8	8	12	1 x12	10 x8	15 x4	6 x0		166
84	12	996156	8	0	8	8	8	8	16 x8	16 x0				184
85	12	996181	8	8	8	8	8	8	16 x8	16 x0				186
86	12	996182	8	4	8	8	8	8	12 x8	16 x4	4 x0			189
87	12	1514326	8	8	8	8	8	8	16 x8	16 x0				191
88	13	8191	6	2	2	6	6	26	1 x26	7 x6	24 x2			9
89	13	69631	6	2	2	10	10	22	1 x22	3 x10	4 x6	24 x2		19
90	13	71679	6	6	6	6	10	22	1 x22	1 x10	10 x6	20 x2		22
91	13	198655	6	2	6	10	14	18	1 x18	1 x14	2 x10	6 x6	22 x2	45
92	13	202239	6	6	6	6	14	18	1 x18	1 x14	12 x6	18 x2		89
93	13	202623	6	6	10	10	10	18	1 x18	3 x10	9 x6	19 x2		51
94	13	218495	6	10	6	6	10	18	1 x18	3 x10	9 x6	19 x2		52
95	13	218751	6	6	6	6	10	18	1 x18	1 x10	15 x6	15 x2		93
96	13	460671	6	6	6	14	14	14	3 x14	1 x10	7 x6	21 x2		74
97	13	463743	6	6	10	10	14	14	2 x14	2 x10	10 x6	18 x2		115
98	13	464759	6	10	10	10	10	14	1 x14	5 x10	7 x6	19 x2		121
99	13	464766	6	6	6	10	10	14	1 x14	3 x10	13 x6	15 x2		127
100	13	471871	6	2	14	10	10	14	2 x14	4 x10	4 x6	22 x2		79
101	13	471903	6	6	10	10	10	14	1 x14	5 x10	7 x6	19 x2		122
102	13	471927	6	6	14	6	10	14	2 x14	2 x10	10 x6	18 x2		116
103	13	471933	6	6	10	6	10	14	1 x14	3 x10	13 x6	15 x2		128
104	13	472431	6	6	6	10	10	14	1 x14	3 x10	13 x6	15 x2		132
105	13	472439	6	10	10	6	10	14	1 x14	3 x10	13 x6	15 x2		169
106	13	996157	6	2	10	10	10	10	6 x10	10 x6	16 x2			173
107	13	996183	6	6	10	10	10	10	6 x10	10 x6	16 x2			174
108	13	996711	6	6	6	10	10	10	4 x10	16 x6	12 x2			180
109	13	1513277	6	10	10	10	10	10	6 x10	10 x6	16 x2			177
110	13	1514301	6	6	10	10	10	10	4 x10	16 x6	12 x2			181
111	13	1514327	6	10	10	10	10	10	6 x10	10 x6	16 x2			-
112	14	16383	4	0	4	4	4	28	1 x28	15 x4	16 x0			6
113	14	73727	4	4	4	8	8	24	1 x24	3 x8	16 x4	12 x0		12
114	14	200703	4	0	4	12	12	20	1 x20	3 x12	12 x4	16 x0		27
115	14	202751	4	4	8	8	12	20	1 x20	1 x12	4 x8	14 x4	12 x0	31
116	14	218623	4	8	4	4	12	20	1 x20	1 x12	4 x8	14 x4	12 x0	32
117	14	218879	4	4	4	4	12	20	1 x20	1 x12	30 x4			85
118	14	219007	4	8	8	8	8	20	1 x20	6 x8	15 x4	10 x0		37
119	14	460799	4	4	4	12	16	16	2 x16	2 x12	14 x4	14 x0		58
120	14	463871	4	4	8	8	16	16	2 x16	4 x8	16 x4	10 x0		98
121	14	464767	4	8	8	12	12	16	1 x16	2 x12	4 x8	14 x4	11 x0	67
122	14	471935	4	4	12	8	12	16	1 x16	2 x12	4 x8	14 x4	11 x0	68
123	14	472447	4	8	8	8	12	16	1 x16	1 x12	6 x8	15 x4	9 x0	102
124	14	472951	4	8	12	8	8	16	1 x16	1 x12	6 x8	15 x4	9 x0	-

Class Number	Group Number	Function Number	s_0	s_1	s_2	s_3	s_4	s_5	Summary Of Complete Spectrum					Book
125	14	472957	4	8	8	8	8	16	1x16	8x8	16x4	7x0		110
126	14	989047	4	8	8	12	12	12	4x12	4x8	12x4	12x0		137
127	14	989054	4	4	4	12	12	12	4x12	28x4				141
128	14	996159	4	0	12	12	12	12	6x12	10x4	16x0			83
129	14	996191	4	4	8	12	12	12	4x12	4x8	12x4	12x0		138
130	14	996221	4	4	8	8	12	12	2x12	8x8	14x4	8x0		156
131	14	996719	4	4	4	12	12	12	4x12	28x4				142
132	14	996727	4	8	8	8	12	12	2x12	8x8	14x4	8x0		160
133	14	1513279	4	8	12	12	12	12	4x12	4x8	12x4	12x0		145
134	14	1513342	4	8	8	8	12	12	2x12	8x8	14x4	8x0		161
135	14	1514303	4	4	12	12	12	12	4x12	28x4				153
136	14	1514335	4	8	8	12	12	12	3x12	6x8	13x4	10x0		149
137	14	1514365	4	8	8	8	12	12	2x12	8x8	14x4	8x0		-
138	14	1515325	4	8	8	12	8	12	2x12	8x8	14x4	8x0		-
139	14	1523005	4	8	8	8	8	12	3x12	6x8	13x4	10x0		150
140	14	1523035	4	8	8	8	8	12	1x12	10x8	15x4	6x0		165
141	14	18290620	4	8	8	8	8	8	12x8	16x4	4x0			188
142	15	32767	2	2	2	2	2	30	1x30	31x2				3
143	15	81919	2	2	6	6	6	26	1x26	7x6	24x2			8
144	15	204799	2	2	6	10	10	22	1x22	3x10	4x6	24x2		18
145	15	219135	2	6	6	6	10	22	1x22	1x10	10x6	20x2		23
146	15	462847	2	2	2	14	14	18	1x18	3x14	28x2			40
147	15	464895	2	6	6	10	14	18	1x18	1x14	2x10	6x6	22x2	43
148	15	472063	2	2	10	6	14	18	1x18	1x14	2x10	6x6	22x2	44
149	15	472575	2	6	6	6	14	18	1x18	1x14	12x6	18x2		88
150	15	472959	2	6	10	10	10	18	1x18	3x10	9x6	19x2		49
151	15	489335	2	10	10	6	6	18	1x18	3x10	9x6	19x2		50
152	15	489339	2	6	10	6	6	18	1x18	1x10	15x6	15x2		92
153	15	989055	2	6	6	14	14	14	3x14	1x10	7x6	21x2		73
154	15	996223	2	2	10	10	14	14	2x14	4x10	4x6	22x2		77
155	15	996735	2	6	6	10	14	14	2x14	2x10	10x6	18x2		113
156	15	997239	2	6	10	10	10	14	1x14	5x10	7x6	19x2		119
157	15	997245	2	6	6	10	10	14	1x14	3x10	13x6	15x2		125
158	15	1513343	2	10	10	10	14	14	2x14	4x10	4x6	22x2		78
159	15	1514367	2	6	10	10	14	14	2x14	2x10	10x6	18x2		114
160	15	1515327	2	6	10	14	10	14	2x14	2x10	10x6	18x2		-
161	15	1515383	2	10	10	10	10	14	1x14	5x10	7x6	19x2		120
162	15	1515390	2	6	6	10	10	14	1x14	3x10	13x6	15x2		126
163	15	1523007	2	6	10	10	10	14	1x14	5x10	7x6	19x2		-
164	15	1523039	2	10	6	10	10	14	1x14	3x10	13x6	15x2		131
165	15	1523070	2	6	6	6	10	14	1x14	3x10	13x6	15x2		-
166	15	2045757	2	6	10	10	10	10	6x10	10x6	16x2			168
167	15	2045783	2	10	10	10	10	10	6x10	10x6	16x2			171
168	15	18290558	2	10	10	10	10	10	6x10	10x6	16x2			172
169	15	18290621	2	10	10	10	10	10	6x10	10x6	16x2			176
170	15	18291671	2	10	10	10	10	10	6x10	10x6	16x2			-
171	15	18291708	2	6	6	6	10	10	4x10	16x6	12x2			179
172	16	65535	0	0	0	0	0	32	1x32	31x0				1
173	16	98303	0	4	4	4	4	28	1x28	15x4	16x0			5
174	16	212991	0	0	8	8	8	24	1x24	7x8	24x0			15
175	16	221183	0	4	4	8	8	24	1x24	3x8	16x4	12x0		11
176	16	466943	0	4	4	12	12	20	1x20	3x12	12x4	16x0		26
177	16	473087	0	4	8	8	12	20	1x20	1x12	4x8	14x4	12x0	30
178	16	489343	0	8	8	8	8	20	1x20	6x8	15x4	10x0		36
179	16	987135	0	0	0	16	16	16	4x16	28x0				55
180	16	989183	0	4	4	12	16	16	2x16	2x12	14x4	14x0		57
181	16	996351	0	0	8	8	16	16	2x16	8x8	22x0			61
182	16	996863	0	4	4	8	16	16	2x16	4x8	16x4	10x0		96
183	16	997247	0	4	8	12	12	16	1x16	2x12	4x8	14x4	11x0	65
184	16	1013623	0	8	8	8	8	16	1x16	12x8	19x0			106
185	16	1013627	0	4	8	8	8	16	1x16	8x8	16x4	7x0		109
186	16	1513471	0	8	8	8	16	16	2x16	8x8	22x0			62
187	16	1514495	0	8	4	8	16	16	2x16	4x8	16x4	10x0		97
188	16	1515391	0	8	8	12	12	16	1x16	2x12	4x8	14x4	11x0	66
189	16	1523071	0	8	8	8	12	16	1x16	1x12	6x8	15x4	9x0	101
190	16	1523581	0	8	8	8	8	16	1x16	12x8	19x0			-
191	16	1523582	0	4	8	8	8	16	1x16	8x8	16x4	7x0		-
192	16	2039671	0	8	8	12	12	12	4x12	4x8	12x4	12x0		135
193	16	2045759	0	4	12	12	12	12	4x12	4x8	12x4	12x0		136
194	16	2045791	0	8	8	12	12	12	3x12	6x8	13x4	10x0		148
195	16	2045821	0	8	8	8	12	12	2x12	8x8	14x4	8x0		155
196	16	18290559	0	12	12	12	12	12	6x12	10x4	16x0			82
197	16	18290623	0	8	12	12	12	12	4x12	4x8	12x4	12x0		144
198	16	18290686	0	8	8	8	12	12	2x12	8x8	14x4	8x0		159
199	16	18291679	0	8	8	12	12	12	3x12	6x8	13x4	10x0		-
200	16	18291709	0	8	8	8	12	12	2x12	8x8	14x4	8x0		-
201	16	18300397	0	8	4	8	8	12	1x12	10x8	15x4	6x0		164

Class Number	Group Number	Function Number	s_0	s_1	s_2	s_3	s_4	s_5	Summary Of Complete Spectrum			Book
202	16	18823036	0	8	8	8	8	8	16 x8	16 x0		183
203	16	18823100	0	4	8	8	8	8	12 x8	16 x4	4 x0	187
204	16	18823126	0	8	8	8	8	8	16 x8	16 x0		185
205	16	54482538	0	0	8	8	8	0	16 x8	16 x0		190

Table 21 – Complete spectral class list for $n = 5$

B.2 Transcription Of Hurst Printouts

This section is a transcription and organization of the most complete data available from the previous works at the time of the writing of this thesis. As indicated earlier, spectral classes listed in Appendix B of [2] are simply summaries of the spectral data, and not complete listings. With only the spectral summary, it is impossible to determine the exact function used as the canonical function. Copies of the original printouts were obtained which provided the complete spectrum for each canonical function. Although the spectral summary in [2] is in its final form, the printouts were not. The printouts simply state the function number listed as the "Hurst Class" in Table 22 and therefore the data "Class Number" column was matched by hand to the summary in [2].

To save space, and to allow more direct comparison between various lists, only the first order coefficients and spectral summary are listed, rather than the entire spectrum. In order to not lose data when displayed in the more compact form, the decimal representation of the function was calculated from the complete spectrum of each function, and added to Table 21.

Class Number§§	Hurst Class***	Function Number†††	s_0	s_1	s_2	s_3	s_4	s_5	Summary Of Complete Spectrum		
1	1a	1771476585	0	0	0	0	0	0	1 x32	31 x0	
2	1b	0	32	0	0	0	0	0	1 x32	31 x0	
3	2a	1771476584	2	-2	-2	-2	-2	-2	1 x30	31 x2	
4	2b	1	30	2	2	2	2	2	1 x30	31 x2	
5	3a	1771476586	0	-4	0	0	0	0	1 x28	15 x4	16 x0
6	3b	1019462463	-4	0	4	4	4	4	1 x28	15 x4	16 x0
7	3c	3	28	0	4	4	4	4	1 x28	15 x4	16 x0

§§ The equivalent class number in [2]

*** As listed on the original printouts

††† Calculated from complete spectra listed on original printouts

Class Number	Hurst Class	Function Number	s_0	s_1	s_2	s_3	s_4	s_5	Summary Of Complete Spectrum				
8	4a	1771476590	-2	-2	-2	2	2	2	1 x26	7 x6	24 x2		
9	4b	267448328	6	-2	-2	-6	-6	-6	1 x26	7 x6	24 x2		
10	4c	7	26	2	2	6	6	6	1 x26	7 x6	24 x2		
11	5b	1019462443	0	-4	4	4	0	0	1 x24	3 x8	16 x4	12 x0	
12	5a	1771476606	-4	0	0	0	4	4	1 x24	3 x8	16 x4	12 x0	
13	5c	16776983	-8	4	4	4	8	8	1 x24	3 x8	16 x4	12 x0	
14	5d	23	24	4	4	4	8	8	1 x24	3 x8	16 x4	12 x0	
15	6a	1771476582	0	0	0	0	0	0	1 x24	7 x8	24 x0		
16	6b	267448320	8	0	0	-8	-8	-8	1 x24	7 x8	24 x0		
17	6c	15	24	0	0	8	8	8	1 x24	7 x8	24 x0		
18	7a	1771476598	-2	2	2	-2	2	2	1 x22	3 x10	4 x6	24 x2	
19	7b	267448336	6	2	2	-10	-6	-6	1 x22	3 x10	4 x6	24 x2	
20	7c	16776991	-10	2	2	6	10	10	1 x22	3 x10	4 x6	24 x2	
21	7d	31	22	2	2	6	10	10	1 x22	3 x10	4 x6	24 x2	
22	8a	1771476862	-6	2	2	2	2	6	1 x22	1 x10	10 x6	20 x2	
23	8b	1019462187	2	-6	2	2	2	-2	1 x22	1 x10	10 x6	20 x2	
24	8c	65256	10	-6	-6	-6	-6	22	1 x22	1 x10	10 x6	20 x2	
25	8d	279	22	6	6	6	6	10	1 x22	1 x10	10 x6	20 x2	
26	9a	1771476566	0	4	0	0	0	0	1 x20	3 x12	12 x4	16 x0	
27	9b	1019462403	4	0	4	4	-4	-4	1 x20	3 x12	12 x4	16 x0	
28	9c	16777023	-12	0	4	4	12	12	1 x20	3 x12	12 x4	16 x0	
29	9d	63	20	0	4	4	12	12	1 x20	3 x12	12 x4	16 x0	
30	10c	1721342073	0	0	0	-4	4	0	1 x20	1 x12	4 x8	14 x4	12 x0
31	10b	267448592	4	4	4	-8	-8	-4	1 x20	1 x12	4 x8	14 x4	12 x0
32	10a	1771476854	-4	4	4	0	0	4	1 x20	1 x12	4 x8	14 x4	12 x0
33	10d	16776735	-8	0	0	4	12	8	1 x20	1 x12	4 x8	14 x4	12 x0
34	10e	65248	12	-4	-4	-8	-8	20	1 x20	1 x12	4 x8	14 x4	12 x0
35	10f	287	20	4	4	8	8	12	1 x20	1 x12	4 x8	14 x4	12 x0
36	11c	267514136	0	4	4	-4	-4	-4	1 x20	6 x8	15 x4	10 x0	
37	11b	1019396651	4	-8	0	0	0	0	1 x20	6 x8	15 x4	10 x0	
38	11a	1771542398	-8	4	4	4	4	4	1 x20	6 x8	15 x4	10 x0	
39	11d	65815	20	8	8	8	8	8	1 x20	6 x8	15 x4	10 x0	
40	12a	1771476502	2	2	2	2	-2	-2	1 x18	3 x14	28 x2		
41	12b	16777087	-14	2	2	2	14	14	1 x18	3 x14	28 x2		
42	12c	127	18	2	2	2	14	14	1 x18	3 x14	28 x2		
43	13a	1771476822	-2	6	2	2	-2	2	1 x18	1 x14	2 x10	6 x6	22 x2
44	13c	267448624	2	2	6	-10	-6	-2	1 x18	1 x14	2 x10	6 x6	22 x2
45	13b	1019462147	6	-2	2	2	-2	-6	1 x18	1 x14	2 x10	6 x6	22 x2
46	13d	16776767	-10	-2	2	2	14	10	1 x18	1 x14	2 x10	6 x6	22 x2
47	13e	65216	14	-2	-6	-6	-10	18	1 x18	1 x14	2 x10	6 x6	22 x2
48	13f	319	18	2	6	6	10	14	1 x18	1 x14	2 x10	6 x6	22 x2
49	14c	1721276537	2	-2	-2	-6	2	2	1 x18	3 x10	9 x6	19 x2	
50	14b	267514128	2	6	6	-6	-6	-6	1 x18	3 x10	9 x6	19 x2	
51	14a	1771542390	-6	6	6	2	2	2	1 x18	3 x10	9 x6	19 x2	
52	14d	16711199	-6	-2	-2	2	10	10	1 x18	3 x10	9 x6	19 x2	
53	14e	130784	10	-2	-2	-6	-6	18	1 x18	3 x10	9 x6	19 x2	
54	14f	65823	18	6	6	10	10	10	1 x18	3 x10	9 x6	19 x2	
55	15a	1771476630	0	0	0	0	0	0	4 x16	28 x0			
56	15b	16777215	-16	0	0	0	16	16	4 x16	28 x0			
57	16a	1771476758	0	4	4	4	-4	0	2 x16	2 x12	14 x4	14 x0	
58	16b	1019462211	4	0	0	0	0	-4	2 x16	2 x12	14 x4	14 x0	
59	16c	16776831	-12	0	0	0	16	12	2 x16	2 x12	14 x4	14 x0	
60	16d	65152	16	-4	-4	-4	-12	16	2 x16	2 x12	14 x4	14 x0	
61	17a	1771476310	0	8	0	0	0	0	2 x16	8 x8	22 x0		
62	17c	267449136	0	0	8	-8	-8	0	2 x16	8 x8	22 x0		
63	17b	1019461635	8	0	0	0	0	-8	2 x16	8 x8	22 x0		
64	17d	64704	16	0	-8	-8	-8	16	2 x16	8 x8	22 x0		
65	18e	1768462249	0	-4	0	0	-4	4	1 x16	2 x12	4 x8	14 x4	11 x0
66	18c	267514160	0	4	8	-8	-4	-4	1 x16	2 x12	4 x8	14 x4	11 x0
67	18a	1771542358	-4	8	4	4	0	0	1 x16	2 x12	4 x8	14 x4	11 x0
68	18f	1010680572	-4	0	-4	-4	8	0	1 x16	2 x12	4 x8	14 x4	11 x0
69	18b	1019396611	8	-4	0	0	-4	-4	1 x16	2 x12	4 x8	14 x4	11 x0
70	18d	16711231	-8	-4	0	0	12	12	1 x16	2 x12	4 x8	14 x4	11 x0
71	18g	130752	12	0	-4	-4	-8	16	1 x16	2 x12	4 x8	14 x4	11 x0
72	18h	65855	16	4	8	8	12	12	1 x16	2 x12	4 x8	14 x4	11 x0

Class Number	Hurst Class	Function Number	s_0	s_1	s_2	s_3	s_4	s_5	Summary Of Complete Spectrum				
73	19a	1771542294	-2	6	6	6	-2	-2	3 x14	1 x10	7 x6	21 x2	
74	19b	1019396675	6	-2	-2	-2	-2	-2	3 x14	1 x10	7 x6	21 x2	
75	19c	16711295	-10	-2	-2	-2	14	14	3 x14	1 x10	7 x6	21 x2	
76	19d	130688	14	-2	-2	-2	-10	14	3 x14	1 x10	7 x6	21 x2	
77	20a	1771541846	-2	10	2	2	2	-2	2 x14	4 x10	4 x6	22 x2	
78	20c	267514672	-2	2	10	-6	-6	-2	2 x14	4 x10	4 x6	22 x2	
79	20d	16710719	-6	-2	-2	-2	14	10	2 x14	4 x10	4 x6	22 x2	
80	20b	1019396099	10	-2	-2	-2	-2	-6	2 x14	4 x10	4 x6	22 x2	
81	20e	130240	14	2	-6	-6	-6	14	2 x14	4 x10	4 x6	22 x2	
82	21a	1771410774	0	12	0	0	0	0	6 x12	10 x4	16 x0		
83	21c	267645744	-4	0	12	-4	-4	-4	6 x12	10 x4	16 x0		
84	21b	1019265027	12	0	-4	-4	-4	-4	6 x12	10 x4	16 x0		
85	22a	1771444094	-4	4	4	4	4	4	1 x20	1 x12	30 x4		
86	22b	32488	12	-4	-4	-4	-4	20	1 x20	1 x12	30 x4		
87	22c	33047	20	4	4	4	4	12	1 x20	1 x12	30 x4		
88	23a	1771472758	-2	2	2	2	2	2	1 x18	1 x14	12 x6	18 x2	
89	23b	267444496	6	2	2	-6	-6	-6	1 x18	1 x14	12 x6	18 x2	
90	23c	61152	14	-6	-6	-6	-6	18	1 x18	1 x14	12 x6	18 x2	
91	23d	4383	18	6	6	6	6	14	1 x18	1 x14	12 x6	18 x2	
92	24b	267481368	2	6	6	-2	-2	-6	1 x18	1 x10	15 x6	15 x2	
93	24a	1771509630	-6	6	6	6	6	2	1 x18	1 x10	15 x6	15 x2	
94	24c	98024	10	-2	-2	-2	-2	18	1 x18	1 x10	15 x6	15 x2	
95	24d	98583	18	6	6	6	6	10	1 x18	1 x10	15 x6	15 x2	
96	25a	1771475798	0	4	4	0	0	0	2 x16	4 x8	16 x4	10 x0	
97	25c	267449648	0	4	4	-8	-8	0	2 x16	4 x8	16 x4	10 x0	
98	25b	1019463171	4	0	0	4	-4	-4	2 x16	4 x8	16 x4	10 x0	
99	25d	16775743	-8	-4	4	0	16	8	2 x16	4 x8	16 x4	10 x0	
100	25e	64192	16	-4	-4	-8	-8	16	2 x16	4 x8	16 x4	10 x0	
101	26c	1771534473	0	0	0	0	0	-4	1 x16	1 x12	6 x8	15 x4	9 x0
102	26a	1771538294	-4	4	4	4	4	0	1 x16	1 x12	6 x8	15 x4	9 x0
103	26d	267460335	-8	0	0	8	8	4	1 x16	1 x12	6 x8	15 x4	9 x0
104	26b	126688	12	-4	-4	-4	-4	16	1 x16	1 x12	6 x8	15 x4	9 x0
105	26e	69919	16	8	8	8	8	12	1 x16	1 x12	6 x8	15 x4	9 x0
106	27b	535949584	0	8	8	-8	-8	-8	1 x16	12 x8	19 x0		
107	27a	2039977846	-8	8	8	0	0	0	1 x16	12 x8	19 x0		
108	27c	268501279	16	8	8	8	8	8	1 x16	12 x8	19 x0		
109	28c	267612440	0	4	8	0	0	-8	1 x16	8 x8	16 x4	7 x0	
110	28a	1771378558	-4	8	4	4	4	4	1 x16	8 x8	16 x4	7 x0	
111	28b	1019232811	8	-4	0	0	0	0	1 x16	8 x8	16 x4	7 x0	
112	28d	229655	16	4	8	8	8	8	1 x16	8 x8	16 x4	7 x0	
113	29a	1771541334	-2	6	6	2	2	-2	2 x14	2 x10	10 x6	18 x2	
114	29c	267515184	-2	6	6	-6	-6	-2	2 x14	2 x10	10 x6	18 x2	
115	29b	1019397635	6	-2	-2	2	-6	-2	2 x14	2 x10	10 x6	18 x2	
116	29d	16710207	-6	-6	2	-2	14	10	2 x14	2 x10	10 x6	18 x2	
117	29e	129728	14	-2	-2	-6	-6	14	2 x14	2 x10	10 x6	18 x2	
118	29f	16647616	10	2	-6	-2	14	-6	2 x14	2 x10	10 x6	18 x2	
119	30a	1754765142	-2	6	2	2	2	2	1 x14	5 x10	7 x6	19 x2	
120	30c	250736944	2	2	6	-10	-2	-2	1 x14	5 x10	7 x6	19 x2	
121	30e	1751672918	6	-2	-6	-6	2	2	1 x14	5 x10	7 x6	19 x2	
122	30b	1036173827	6	-2	2	2	-6	-6	1 x14	5 x10	7 x6	19 x2	
123	30d	33488447	-10	-2	2	2	10	10	1 x14	5 x10	7 x6	19 x2	
124	30f	16843071	14	6	10	10	10	10	1 x14	5 x10	7 x6	19 x2	
125	**31a**								**1 x14**	**3 x10**	**13 x6**	**15 x2**	
126	**31b**								**1 x14**	**3 x10**	**13 x6**	**15 x2**	
127	**31c**								**1 x14**	**3 x10**	**13 x6**	**15 x2**	
128	**31d**								**1 x14**	**3 x10**	**13 x6**	**15 x2**	
129	**31e**								**1 x14**	**3 x10**	**13 x6**	**15 x2**	
130	**31f**	**198999**	**14**	**6**	**6**	**10**	**10**	**10**	**1 x14**	**3 x10**	**13 x6**	**15 x2**	
131	32a	697800534	-2	6	6	6	2	2	1 x14	3 x10	13 x6	15 x2	
132	32b	2093138435	6	-2	-2	-2	-6	-6	1 x14	3 x10	13 x6	15 x2	
133	32c	1090453055	-10	-2	-2	-2	10	10	1 x14	3 x10	13 x6	15 x2	
134	32d	1073807679	14	6	6	6	10	10	1 x14	3 x10	13 x6	15 x2	
135	33a	1771279702	0	8	4	0	0	0	4 x12	4 x8	12 x4	12 x0	
136	33e	868863756	0	0	-4	8	-8	0	4 x12	4 x8	12 x4	12 x0	
137	33c	1520477797	4	-4	0	-4	-4	4	4 x12	4 x8	12 x4	12 x0	

Class Number	Hurst Class	Function Number	s_0	s_1	s_2	s_3	s_4	s_5	Summary Of Complete Spectrum			
138	33d	267776816	-4	4	8	-4	-4	-4	4 x12	4 x8	12 x4	12 x0
139	33b	1019658243	8	0	-4	0	0	-8	4 x12	4 x8	12 x4	12 x0
140	33f	392384	12	4	-8	-4	-4	12	4 x12	4 x8	12 x4	12 x0
141	34a	1771410422	-4	4	4	-4	4	4	4 x12	28 x4		
142	34b	267646352	-4	4	4	-4	-4	-4	4 x12	28 x4		
143	34c	260704	12	-4	4	-4	-4	12	4 x12	28 x4		
144	35a	1754765078	0	4	4	4	0	0	4 x12	4 x8	12 x4	12 x0
145	35b	1036173891	4	0	0	0	-4	-4	4 x12	4 x8	12 x4	12 x0
146	35d	1751672854	8	-4	-4	-4	0	0	4 x12	4 x8	12 x4	12 x0
147	35c	33488511	-12	0	0	0	12	12	4 x12	4 x8	12 x4	12 x0
148	36a	1770493238	0	4	4	4	0	0	3 x12	6 x8	13 x4	10 x0
149	36b	1020444771	4	0	0	-8	4	-4	3 x12	6 x8	13 x4	10 x0
150	36d	15662175	-4	0	-8	0	12	12	3 x12	6 x8	13 x4	10 x0
151	36c	870174572	-8	4	-4	4	0	0	3 x12	6 x8	13 x4	10 x0
152	36e	1178784	12	0	0	-8	-4	12	3 x12	6 x8	13 x4	10 x0
153	37a	697800662	-4	4	4	4	4	4	4 x12	28 x4		
154	37b	1090453183	-12	-4	-4	-4	12	12	4 x12	28 x4		
155	38b	1019662507	0	-4	-4	-4	4	0	2 x12	8 x8	14 x4	8 x0
156	38a	1771275774	-4	0	0	0	8	4	2 x12	8 x8	14 x4	8 x0
157	38c	1768719617	8	4	-4	4	-4	-8	2 x12	8 x8	14 x4	8 x0
158	38d	388200	12	0	-8	0	0	12	2 x12	8 x8	14 x4	8 x0
159	39a	697800022	0	8	4	4	4	0	2 x12	8 x8	14 x4	8 x0
160	39c	446998117	4	-4	0	0	0	4	2 x12	8 x8	14 x4	8 x0
161	39d	1341256496	-4	4	8	-8	-8	-4	2 x12	8 x8	14 x4	8 x0
162	39b	2093137923	8	0	-4	-4	-4	-8	2 x12	8 x8	14 x4	8 x0
163	39e	1073872064	12	4	-8	-8	-8	12	2 x12	8 x8	14 x4	8 x0
164	40c	1010557082	0	-8	4	-4	4	-4	1 x12	10 x8	15 x4	6 x0
165	40a	1771156784	4	4	8	-8	-8	0	1 x12	10 x8	15 x4	6 x0
166	40b	1520600579	8	-8	4	4	-4	-4	1 x12	10 x8	15 x4	6 x0
167	40d	467801	12	4	8	8	8	8	1 x12	10 x8	15 x4	6 x0
168	41a	1753716022	2	2	2	2	2	2	6 x10	10 x6	16 x2	
169	41b	853397356	-6	2	-6	2	2	2	6 x10	10 x6	16 x2	
170	41c	17956000	10	2	2	-6	-6	10	6 x10	10 x6	16 x2	
171	42a	697997878	2	-10	6	2	2	2	6 x10	10 x6	16 x2	
172	42e	2083573660	-2	2	-6	-2	-10	6	6 x10	10 x6	16 x2	
173	42c	447326469	6	-6	2	6	-2	-2	6 x10	10 x6	16 x2	
174	42d	1341979728	-6	6	-10	-6	-6	-6	6 x10	10 x6	16 x2	
175	42b	2093993827	-10	2	-6	6	-2	-2	6 x10	10 x6	16 x2	
176	43a	697668950	2	10	2	2	2	2	6 x10	10 x6	16 x2	
177	43c	1341387568	-6	2	10	-6	-6	-6	6 x10	10 x6	16 x2	
178	43b	2093006851	10	2	-6	-6	-6	-6	6 x10	10 x6	16 x2	
179	44c	1989188697	2	6	-2	-6	-6	2	4 x10	16 x6	12 x2	
180	44a	2040501078	-6	6	6	2	2	-6	4 x10	16 x6	12 x2	
181	44b	536474928	-6	6	6	-6	-6	-6	4 x10	16 x6	12 x2	
182	44d	269089472	10	-2	-2	-6	-6	10	4 x10	16 x6	12 x2	
183	**45a**	**1788184500**	**0**	**0**	**0**	**0**	**0**	**0**	**16 x8**	**16 x0**		
184	**45a**	**818929134**	**-8**	**0**	**-8**	**0**	**8**	**8**	**16 x8**	**16 x0**		
185	**45b**								**16 x8**	**16 x0**		
186	**45b**								**16 x8**	**16 x0**		
187	46c	1674235660	0	-4	-8	8	-8	0	12 x8	16 x4	4 x0	
188	46a	965776726	4	8	4	-4	-4	4	12 x8	16 x4	4 x0	
189	46b	1824899075	8	4	-8	0	0	-8	12 x8	16 x4	4 x0	
190	48b	1522983501	0	0	-8	0	0	0	16 x8	16 x0		
191	48a	1777717630	-8	8	0	-8	8	0	16 x8	16 x0		

Table 22 – Transcription of Hurst Printouts

Hurst classes 31 and 45 have been highlighted in Table 21 as the complete data is unavailable. The function number for class 130 (Hurst class 31f) comes directly from the Hurst function number listed in Figure 36 for case 30. This function was cross-referenced with the complete data from the implementation in Appendix A that was determined to be in spectral class 32 (from Table 21), which is equivalent to class 130 in [2]. The listed spectral summary is based on the data from the implementation in Appendix A.

The Hurst class 45 should in fact be 2 separate classes, as discussed in [2], but the printouts did not reflect this. As there is no additional information available on which functions are listed for Hurst class 45, they are assumed to be class 45a and that 45b is the missing data. It is fairly certain that the functions listed for Hurst class 45 in Table 22 are in the correct Hurst class, but it is uncertain which functions correspond to sub-class.

```
CLASS    HURST    PRIMARY          SUMMARY
         CLASS    COEFFICIENTS

   1     31A      13 7 5 5 5 3     1x13, 1x7, 3x5, 12x3, 15x1
   2     31B      15 7 5 5 5 3     1x15, 1x7, 3x5, 13x3, 14x1
   3     31C      13 7 5 5 3 3     1x13, 1x7, 3x5, 12x3, 15x1
   4     31D      15 7 5 5 3 3     1x15, 1x7, 3x5, 13x3, 14x1
   5     31E      11 7 5 5 3 3     1x11, 1x7, 2x5, 13x3, 15x1
   6     31F      9 5 5 5 3 3      1x9, 3x5, 13x3, 15x1


CASE 31
-------

FUNCTION 00030957

IN BINARY 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 0 1 0 1 0 1 0 1 1 1

SPECTRUM  31A       13 -1 -1 3 -3 1 -3 1 1 1 1 1 -1 -1 3 -5 -3 -3 1 1 3 3 3
-------------
TRANSFORMED SPECTRUM
13 -3 -5 -3 -5 -3 -1 1 -5 1 3 1 -1 -3 3 1 -7 3 -3 -1 1 3 1 3 -3 3 1 -1 1 -1

SPECTRUM  31B       15 3 3 -1 1 -3 1 -3 1 1 1 1 -1 -1 -5 3 -3 -3 1 1 3 3 -5
-------------
TRANSFORMED SPECTRUM
15 3 -5 -1 -5 3 -1 -1 -5 3 3 3 3 -1 3 -1 -7 -3 1 -3 1 1 1 1 -3 -3 1 1 1 -3

SPECTRUM  31C       13 -1 -1 3 1 -3 1 -3 1 1 1 1 -5 3 -1 -1 1 1 -3 -3 -5 3 3
-------------
TRANSFORMED SPECTRUM
13 -3 3 1 -5 1 3 -3 -5 1 -1 1 -5 -3 -1 -3 -7 -1 1 3 1 3 1 -1 1 3 -3 3 1 -1

SPECTRUM  31D       15 -3 1 -3 3 -1 -1 3 -5 3 -1 -1 1 1 1 1 -5 3 3 3 1 1 -3
-------------
TRANSFORMED SPECTRUM
15 -3 -3 1 -5 -1 -1 3 -5 -1 3 -1 1 -3 1 -3 -7 1 -3 -3 3 3 -1 -1 -5 3 3 3 1

SPECTRUM  31E       11 3 -1 -1 1 1 1 1 1 -3 1 -3 3 -1 -1 3 -7 -3 -3 1 -5 -1
-------------
TRANSFORMED SPECTRUM
11 -3 3 1 -5 1 3 -3 -5 1 -1 1 3 -3 -1 -3 -7 -1 -3 -1 1 3 -3 3 1 3 1 -1 1 -1

SPECTRUM  31F       9 -3 -3 1 -5 -1 3 -1 -5 3 3 3 1 1 -3 -3 -5 3 -1 -1 1 1 1
-------------
```

Figure 36 – Addendum printout for Hurst Class 31 in 0/1 encoding

The Hurst class 47 is not listed in Table 22 because the functions listed on the printouts were determined to be invalid functions. As expected, since these functions are invalid, they do not correspond with any functions listed in [2]. The data provided in an Addendum printout, as seen in the Figure 36 transcription, is incomplete due to wide-format printer paper being photocopied onto letter sized paper in a portrait orientation. The original copy is unavailable.

B.3 Summary

The tables in this chapter provide a complete list of the spectral classes, and their canonical functions, produced in this research. Additionally, a reconstructed table of the data used to generate the signatures tabulated in [2] is provided. Although 100% of the data used for [2] could not be recovered, a large percentage of the information could be reconstructed using the tables in [2] and intermediate data used for the publication.

The list of spectral classes tabulated from this research should provide adequate data for use for comparison in future work as the canonical functions are presented in a form that allows complete reconstruction of the spectral coefficients, and provides and exact listing of the chosen canonical function for each class.

Appendix
C - Source Code

C.1 Main

C.1.1 main.h

```
/*
 *  main.h
 *  Classify
 *
 *  Created by Neil Anderson on 2006-10-20.
 *  Copyright 2006 Neil Anderson. All rights reserved.
 *
 */

#include <iostream>
#include <vector>
#include <map>
#include <algorithm>
#include <cmath>
#include <fstream>
#include <string>
#include <cstdlib>
#include <sstream>

using namespace std;
```

C.1.2 main.cpp

```
/*
 *  main.cpp
 *  Classify
 *
 *  Created by Neil Anderson on 2006-08-10.
 *  Copyright 2006 Neil Anderson. All rights reserved.
 *
 */

#include "main.h"
#include "Prefilter.h"
#include "Classify.h"
#include "Misc.h"

int main (int argc, char * const argv[]) {
   // -- Various values used throuout --
   // Rather than passing these values throughout the program they are
   // just re-calculated in a few places
   int numVar = 5;
   int vecLen = pow(2.0, (double)numVar);
   unsigned int numFn = (pow(2.0, (double)vecLen))/2;
   // ---------------------------------

   // -- Array that stores the number of items in each group --
   // also used for pre-filter temp file names
   int *len;
   len = new int[vecLen];

   int numGroups = vecLen/2;

   for (int i = 0; i < numGroups; i++) {
      // binomial coefficient
      len[i] = (int)C(vecLen, i+1);

      // the last group only needs to store half
      if (i == numGroups-1)
         len[i] /= 2;
   }
   // -----------------------------------------------------

   // -- Prefilter the groups --
   // These lines can be commented out if we want to use temp files
   // that have previously been generated (they have to be placed in
   // the running director).
   Prefilter p(numVar);
   p.pre(len);
   // -------------------------

   // -- Classify the data contained in the temp files --
   // The return value of classes (the data it points to) used to be printed
   // out at the end, but now it is printed within the generateClasses
   // method in order to print the results after each group is processed
   // in case the program is stopped part way through...then at last you get
   // partial results.
   Classify c(numVar);
   unsigned char *classes;
   classes = c.generateClasses(len);
   // -----------------------------------------------

   // -- Print the 0 case ----------
   if (fileExists("output0"))
      remove("output0");

   fstream fout("output0", ios::out | ios::app);
   fout << "f(0)\tclass: 0" << endl;
   fout.close();
   // -----------------------------

   return 0;
}
```

C.2 Prefilter

C.2.1 Prefilter.h

```
/*
 *  Prefilter.h
 *  Classify
 *
 *  Created by Neil Anderson on 2006-10-20.
 *  Copyright 2006 Neil Anderson. All rights reserved.
 *
 */

#include "main.h"

class Prefilter {
private:
   unsigned int numVar, vecLen, numFn;

   int *destList;
public:
   Prefilter(int);
   ~Prefilter();
   void pre(int *);
};
```

C.2.2 Prefilter.cpp

```
/*
 *  Prefilter.cpp
 *  Classify
 *
 *  Created by Neil Anderson on 2006-10-20.
 *  Copyright 2006 Neil Anderson. All rights reserved.
 *
 */

#include "Prefilter.h"
#include "Misc.h"

Prefilter::Prefilter(int num) {
   numVar = num;
   vecLen = pow(2.0, (double)numVar);
   numFn = (pow(2.0, (double)vecLen))/2;
}

// Deconstructor cleans up the temp files so there's no crap laying around.
Prefilter::~Prefilter() {
   int s = vecLen/2;

   for (int i = 0; i < s; i++) {
      remove(intToString(destList[i]).c_str());
   }
}

void Prefilter::pre(int *len) {
   // -- Create vector of temp files --
   vector<fstream*> g;

   destList = len;

   int numGroups = vecLen/2;

   // Files are set to write-only
   for (int i = 0; i < numGroups; i++) {
      // Since we are appending files, we want to make sure we're
      // starting clean - remove any file with the same name we want to use
      // if it already exists.
      if (fileExists(intToString(len[i])))
```

```
         remove(intToString(len[i]).c_str());

      g.push_back(new fstream(intToString(len[i]).c_str(), ios::out | ios::app));
   }
   // ---------------------------------

   for (unsigned int i = 0; i < numFn; i++) {
      // -- Count the number if true bits --
      unsigned int res = 0;
      for (unsigned int j = 0; j < vecLen; j++) {
         res += (i >> j) & 1;
      }
      // -----------------------------------

      for (int j = 1; j <= numGroups; j++) {
         // Assign the value to the correct file. Sort by the number
         // of true bits in the integer.
         // Ex. 00110001 has 3 true bits. All integers with 3 true or 3 false
         // bits should be in the same category.
         if (res == j || res == vecLen - j) {
            (*g[j-1]) << i << endl;
            break;
         }
      }
   }

   // -- Close all the files we created and had open --
   for (int i = 0; i < g.size(); i++) {
      (*g[i]).close();
   }
   // -------------------------------------------------
}
```

C.3 Rules

C.3.1 Rules.h

```
/*
 *  Rules.h
 *  ClassifyCPP
 *
 *  Created by Neil Anderson on 2006-08-05.
 *  Copyright 2006 Neil Anderson. All rights reserved.
 *
 */

#include "main.h"
#include "Misc.h"

class Rules {
private:
   vector<vector<int> > t1Rules;
   vector<vector<int> > t2Rules;
   vector<vector<int> > t4Rules;
   vector<vector<int> > t4List;

   int N; // same as vecLen
   int len; // same as numVar

   void permuteVarRules(int);
   void negateVarRules(int);
   void typeFourRules(int);
   void genTypeFourList(int);

public:

   Rules(int);

   void generateRules(void);

   void printRules(void);

   vector<vector<int> > getType1(void);
   vector<vector<int> > getType2(void);
   vector<vector<int> > getType4(void);

   void getType1Arr(int **, int, int);
   void getType2Arr(int **, int, int);
   void getType4Arr(int **, int, int);
};
```

C.3.2 Rules.cpp

```
/*
 *  Rules.cpp
 *  ClassifyCPP
 *
 *  Created by Neil Anderson on 2006-08-05.
 *  Copyright 2006 Neil Anderson. All rights reserved.
 *
 */

#include "Rules.h"

Rules::Rules(int numVar) {
   len = numVar;
   // N is the same as vecLen
   N = pow(2.0, (double) numVar);
}

void Rules::generateRules() {
   permuteVarRules(N);
   negateVarRules(N);
   typeFourRules(N);
}

// Type 1 Rules accessor method
vector<vector<int> > Rules::getType1() {
   return t1Rules;
}

// Type 2 Rules accessor method
vector<vector<int> > Rules::getType2() {
   return t2Rules;
}

// Type 3 Rules accessor method
vector<vector<int> > Rules::getType4() {
   return t4Rules;
}
```

```
// Convert Type 1 Rules vector to 2D array
// Arrays are faster to access than vectors. This matters because
// they are accessed so many times in this program.
void Rules::getType1Arr(int **tmp, int h, int w) {
   for (int i = 0; i < h; i++) {
      for (int j = 0; j < w; j++) {
         tmp[i][j] = t1Rules[i][j];
      }
   }
}

// Convert Type 2 Rules vector to 2D array
// Arrays are faster to access than vectors. This matters because
// they are accessed so many times in this program.
void Rules::getType2Arr(int **tmp, int h, int w) {
   for (int i = 0; i < h; i++) {
      for (int j = 0; j < w; j++) {
         tmp[i][j] = t2Rules[i][j];
      }
   }
}

// Convert Type 2 Rules vector to 2D array
// Arrays are faster to access than vectors. This matters because
// they are accessed so many times in this program.
void Rules::getType4Arr(int **tmp, int h, int w) {
   for (int i = 0; i < h; i++) {
      for (int j = 0; j < w; j++) {
         tmp[i][j] = t4Rules[i][j];
      }
   }
}

// Generate Type 1 Rules
void Rules::permuteVarRules(int num_pos) {
   vector<int> intArr;
   vector<int>::iterator iterBegin;
   vector<int>::iterator iterEnd;

   // initialize the vector to [0][1] ... [n-1][n]
   for (int i = 0; i < len; i++) {
      intArr.push_back(i);
   }

   // -- Pointers to the beginning and end of the vector --
   iterBegin = intArr.begin();
   iterEnd = intArr.end();
   // ----------------------------------------------------

   do {
      vector<int> a_rule;
      for (int j = 0; j < num_pos; j++) {
         // convert the function into a bit vector
         vector<int> org_term = itobv(j, len);
         vector<int> new_term(len);

         int s = org_term.size();
         // assign the re-arranged bits to the new term
         for (int k = 0; k < s; k++) {
            new_term[k] = org_term[intArr[k]];
         }
         // convert the new term back to an in
         int index = bvtoi(new_term);
         // add the order value to the order list
         a_rule.push_back(index);
      }
      // Add the rule to the list
      t1Rules.push_back(a_rule);

   // Permute the values until there are no new permutations
   } while (next_permutation(iterBegin, iterEnd));
}

// Generate the type 2 rules
void Rules::negateVarRules(int num_pos) {
   for (int j = 0; j < num_pos; j++) {
      vector<int> a_rule;
      for (int i = 0; i < num_pos; i++) {
         // XOR between numbers 0 - 2^n and all the truth table
         // entries (represented as integer...which also happens
         // to be 0 - 2^n)
         int index = i ^ j;
         a_rule.push_back(index);
      }
      t2Rules.push_back(a_rule);
   }
}

// Generate the type 4 rules
void Rules::typeFourRules(int num_pos) {
   vector<int> initOrder;

   // Add the unmodified version to the list (we need an unswapped
   // function for each step).
   for (int i = 0; i < num_pos; i++) {
      initOrder.push_back(i);
   }

   t4Rules.push_back(initOrder);

   // Generate the list of combinations
   genTypeFourList(num_pos);

   int ts = t4List.size();
   for (int i = 0; i < ts; i++) {
      vector<int> a_rule;
      for (int k = 0; k < num_pos; k++) {
         vector<int> tmp;
         for (int m = 0; m < len; m++) {
            tmp.push_back(0); // entry placeholder with 0 value
            for (int j = 0; j < len; j++) {
               // Take each entry from the t4 list (combinations of xored vars)
               // and multiplies each bit in the entry by that variable's
               // assigned value (k)
               // Set entry to correct value:
               tmp[m] = tmp[m] ^ (((t4List[i][m] >> j) & 1) * ((k >> j) & 1));
               // result:
               // tmp[0] = (x*x's value) ^ (y*y's value) ^ ...
               // depending on what bits are "used" in the mth variable of the
               // ith entry of the t4 list
            }
         }
         int tres = 0;

         // Turn the vector into an int (could use the newly built
         // bvtoi method instead, but this was written before and it
         // works, so why fix it?)
         for (int c = 0; c < len; c++) {
            tres = (tres << 1) | tmp[c];
         }
         // add the value to the rule
         a_rule.push_back(tres);
      }
      // add the rule to the list
      t4Rules.push_back(a_rule);
   } // end for the size of the t4List
}

// The method that derives a list of all possible valid input combinations.
// This list is used for calculating the final type for rules.
void Rules::genTypeFourList(int num_pos) {
   int arrHeight = len;
   int arrWidth = N/2;

   // This is the lookup table for all possible combinations
   // to be calculated from. Rather than hard-coding it, this
   // is calculated once at runtime.
   vector<vector<int> > arr;

   // Initialize the vector for the given size (based on values passed in
   // at runtime.
   for (int i = 0; i < arrHeight; i++) {
      arr.push_back( *(new vector<int>) );
      for (int j = 0; j < arrWidth; j++) {
         arr[i].push_back(0);
      }
   }

   // Assign the values for the "A" row as odd numbers beginning at 1
   // so: 1, 3, 5, 7, 9, ...
   int skip = 0;
   for (int i = 0; i < arrWidth; i++) {
      arr[0][i] = (i + 1) + skip;
      skip++;
   }

   // Build the remaining rows based on "previous row"
   for (int i = 1; i < arrHeight; i++) {
      skip = pow(2.0, (double) (i - 1));
      for (int j = 0; j < arrWidth; ) {
         for (int k = 0; k < skip; k++) {
            arr[i][j] = arr[i-1][j] + skip;
            j++;
         }
         for (int k = 0; k < skip; k++) {
            arr[i][j] = arr[i-1][j];
            j++;
         }
      }
   }
```

```
// In concept we create a temp 2d vector (the data is 4 cells wide...one
// for each variable) and copy item from each row of the lookup table in
// every possible combination. Not all of these combinations are valid as
// we can't have the same variables from different rows. [a][b][c^a][d^a]
// would be valid, while [a^c][b][c^a][d^a] is not (a^c occurs twice...
// c^a - or 0101 - can be wruitten as a^c - 0101). In reality, we do the
// checking for validity in the same step as the list creation to avoid
// storing a huge temporary table.

int numCol = arrWidth;
int numRow = arrHeight;
int endAt = pow((double) numCol, (double) numRow);
int vl = numRow;

// Creat a vector of size n where n is the number of variables.
// Keeps track of which items from each row we are using. To
// ensure we achieve ever combination.
vector<int> vars(vl);

for (int i = 0; i < endAt; i++) {
   vector<int> tmp;
   // increment the appropriate row based on what was added to the
   // working tmp vector.
   for (int j = vl - 2; j >= 0; j--) {
      if (vars[j+1] == numCol) {
         vars[j+1] = 0;
         vars[j]++;
      }
   }

   // get the next combination
   for (int j = 0; j < vl; j++) {
      tmp.push_back(arr[j][vars[j]]);
   }

   vars[vl-1]++;

   vector<int> a(N);

   // Calculate the linear independence of this combination
   // First run
   for (int m = 1; m < N; m++) {
      a[m] = tmp[0] * ((m >> (numRow-1)) & 1);
   }

   // For all the consecutive calcs.
   for (int m = 1; m < N; m++) {
      for (int k = 1; k < numRow; k++) {
         a[m] ^= tmp[k] * ((m >> (numRow-k-1)) & 1);
      }
   }

   int aRes = 1;

   int al = a.size();

   // Check if it is independent or not. If it is, add it to the list
   // of known valid combinations
   for (int m = 1; m < al; m++) {
         if (a[m] == 0)
            aRes = 0;
      }

      if (aRes) {
         t4List.push_back(tmp);
      }
   }
}

// Debug method for printing out the contents
// of each rule vector
void Rules::printRules() {
   cout << endl << "Type 1" << endl;
   for (int i = 0; i < t1Rules.size(); i++) {
      cout << i << "\t";
      for (int j = 0; j < t1Rules[i].size(); j++) {
         cout << t1Rules[i][j] << " ";
      }
      cout << endl;
   }

   cout << endl << "Type 2" << endl;
   for (int i = 0; i < t2Rules.size(); i++) {
      cout << i << "\t";
      for (int j = 0; j < t2Rules[i].size(); j++) {
         cout << t2Rules[i][j] << " ";
      }
      cout << endl;
   }

   cout << endl << "Type 4" << endl;
   for (int i = 0; i < t4Rules.size(); i++) {
      cout << i << "\t";
      for (int j = 0; j < t4Rules[i].size(); j++) {
         cout << t4Rules[i][j] << " ";
      }
      cout << endl;
   }
}
```

C.4 Classify

C.4.1 Classify.h

```
/*
 *  Classify.h
 *  ClassifyCPP
 *
 *  Created by Neil Anderson on 2006-08-05.
 *  Copyright 2006 Neil Anderson. All rights reserved.
 *
 */

#include "main.h"

class Classify {
private:

   int numVar, vecLen;
   unsigned int numFn;

   // -- 2D arrays of rules --
   int **type1;
   int **type2;
   int **type4;
   // -----------------------

   // -- Length of the rule arrays --
   int s1, s2, s4;

public:
   Classify(int);

   unsigned char *generateClasses(int*);
};
```

C.4.2 Classify.cpp

```
/*
 *  Classify.cpp
 *  ClassifyCPP
 *
 *  Created by Neil Anderson on 2006-08-05.
 *  Copyright 2006 Neil Anderson. All rights reserved.
 *
 */

#include "Classify.h"
#include "Rules.h"
#include "Transform.h"

Classify::Classify(int num) {
   numVar = num;
   vecLen = pow(2.0, (double)numVar);
   numFn = (pow(2.0, (double)vecLen))/2;

   // -- Generate the rules --
   Rules r(numVar);
   r.generateRules();
   // -----------------------

   // -- Save the size of each array --
   s1 = (r.getType1()).size();
   s2 = (r.getType2()).size();
   s4 = (r.getType4()).size();
   // --------------------------------

   // -- Allocate a 2D array based on the size of the generated vector --
   type1 = allocArr(s1, vecLen);
   type2 = allocArr(s2, vecLen);
   type4 = allocArr(s4, vecLen);
   // -------------------------------------------------------------------

   // -- Create an array version of the array vector --
   // The array has much faster accesses than a vector (which matters for
   // millions of accesses)
   r.getType1Arr(type1, s1, vecLen);
   r.getType2Arr(type2, s2, vecLen);
   r.getType4Arr(type4, s4, vecLen);
   // ------------------------------------------------
```

```
}

unsigned char *Classify::generateClasses(int *len) {
   // -- Vector of temp files --
   vector<fstream*> g;

   int numGroups = vecLen/2;

   // Files are set to read-only
   for (int i = 0; i < numGroups; i++) {
      g.push_back(new fstream(intToString(len[i]).c_str(), ios::in));
   }

   // Store size of the vector (faster than computing each time)
   int gSize = g.size();
   // --------------------------


   // Transform contructor
   Transform t(numVar, type1, type2, type4, s1, s2, s4);

   // -- Create array for results --
   unsigned char *arr;
   arr = new unsigned char[numFn];
   // ------------------------------

   int t2Flag = 0;
   // -- Keep track of class number --
   int count = 0; // store the highest assigned class number "so far"
   int fl = 0;
   // --------------------------------
   string line = "";

   // For each file in the vector
   for (int i = 0; i < gSize; i++) {

      // Initialize the array to an invalid value (so we can tell what
      // values have been found, and which have been not). Since 0
      // is a special case, we know none of the other values will ever
      // be 0. Originally this was -1, but an unsigned bool cannot store
      // a -1.
      for (unsigned int h = 0; h < numFn; h++) {
         arr[h] = 0;
      }

      // Originally the flag was set so we'd only do negation of output
      // on the last category, but this isn't correct so it is ignored
      // in the transform code. Not used and can be removed (but since it works
      // I'm not going to mess with it).
      if (i == (gSize - 1))
         t2Flag = 1;

      bool *tmpArr; // results from the transformation call
      int lineInt;

      // get the first line
      getline((*g[i]), line);
      while(!(*g[i]).eof()) {
         // turn the string into an int
         lineInt = atoi(line.c_str());
```

```
count++;
fl = 0;

// Check to see if we've already processed this entry. If we have,
// skip it because it's useless extra processing. This speeds up
// execution time dramatically.
if (arr[lineInt] == 0) {

   // Do the transformation on this function and store a pointer
   // to the results.
   tmpArr = t.trans(lineInt, t2Flag);

   // -- Fold the results into the main array --
   // This will hold all the results to-date for this
   // particular category (temp file)

   // Since we are processing this function, there's a chance
   // that the increment in class number is valid.
   fl = 1;

   // this might be the right class number, so assign it for now.
   int classNum = count;

   for (unsigned int j = 0; j < numFn; j++) {
      // if the function wasn't found, skip past the rest of the
      // code in the for loop
      if (tmpArr[j] != 1)
         continue;

      // if the function has already been classified, use that
      // class number instead.
      if (arr[j] != 0) {
         classNum = arr[j];
         // since we already have a class number, the new one in
         // "count" won't be valid, so flag it.
         fl = 0;
         // Break because once we find one, ALL found functions will
         // be the same class, so there's no point continuing.
         break;
      }
   }

   // Assign the class number for all of the found functions
   for (unsigned int m = 0; m < numFn; m++) {
      if (tmpArr[m] == 1) {
         arr[m] = classNum;
      }
   }
   // -----------------------------------------

   // clean up
   delete[] tmpArr;
}
// If the flag is not true, we shouldn't have incremented out
// class number, so correct it.
if (fl == 0)
   count--;

// Get the next line
```

```
         getline((*g[i]), line);
      } // end while file is not EOF

      // Close the file now that it's been finished.
      (*g[i]).close();

      // -- Output the results for this pre-filtered file --
      // File name
      string s = "output" + intToString(i+1);

      // Check if old files are laying around. Remove it if there is.
      if (fileExists(s.c_str()))
         remove(s.c_str());

      fstream fout(s.c_str(), ios::out | ios::app);

      for (unsigned int p = 0; p < numFn; p++) {
         // Only print if there's a valid entry (non-0)
         if (arr[p] != 0)
            fout << "f("<< p << ")\tclass: " << (int)arr[p] << endl;
      }

      fout.close();
      // --------------------------------------------------

   } // end for each file

   // Return a pointer to the array of results
   // (no longer useful since the printing is now done within
   // this method to print out the results "to date")
   return arr;
}
```

C.5 Transform

C.5.1 Transform.h

```
/*
 *  Transform.h
 *  ClassifyCPP
 *
 *  Created by Neil Anderson on 2006-08-05.
 *  Copyright 2006 Neil Anderson. All rights reserved.
 *
 */

#include "main.h"

class Transform {
private:
   void permuteVar(unsigned int, int);
   void negateVar(unsigned int, int);
   void negateOut(unsigned int, int);
   void typeFour(unsigned int, int);
   void addFunction(unsigned int);

   unsigned int swapBits(unsigned int, int*);

   bool *arr;
   unsigned int mask;
   int numVar, vecLen;
   unsigned numFn;
   int s1, s2, s4;

   int **type1;
   int **type2;
   int **type4;

   int *a;

public:
   Transform(int, int**, int**, int**, int, int, int);
   ~Transform();

   bool *trans(unsigned int, int);
};
```

C.5.2 Transform.cpp

```
/*
 *  Transform.cpp
 *  ClassifyCPP
 *
 *  Created by Neil Anderson on 2006-08-05.
 *  Copyright 2006 Neil Anderson. All rights reserved.
 *
 */

#include "Transform.h"

Transform::Transform (int num, int **t1, int **t2, int **t4, int size1, int size2, int
size4) {

   numVar = num;
   vecLen = pow(2.0, (double)numVar);
   numFn = (pow(2.0, (double)vecLen))/2;

   // -- Assign passed values to global vars --
   type1 = t1;
   type2 = t2;
   type4 = t4;

   s1 = size1;
   s2 = size2;
   s4 = size4;
   // ----------------------------------------

   a = new int[vecLen];

   // -- Calculate the mask for inverted output --
   for (int i = 0; i < vecLen; i++) {
      mask = (mask << 1) | 1;
   }
   // --------------------------------------------

}

// Cleanup
Transform::~Transform() {
   delete[] a;
}

bool *Transform::trans(unsigned int num, int t2Flag = 0) {
   arr = new bool[numFn];

   // Initialize to false
   for (unsigned int j = 0; j < numFn; j++) {
      arr[j] = 0;
   }

   // Apply Type 1 tranformation
   permuteVar(num, t2Flag);

   return arr;
}

// Type 1 Transfomation
// This method transforms the function into all derivative functions
// by applying the Type 1 rules to the function passed in as a parameter
void Transform::permuteVar(unsigned int num, int t2Flag) {
   unsigned int newNum;

   // For each item in the Type 1 rule list: swap the bits and pass
   // this new function to the Type 2 transformation method
   for (int i = 0; i < s1; i++) {
      newNum = swapBits(num, type1[i]);
      negateVar(newNum, t2Flag);
   }

}

// Type 2 Transfomation
// This method transforms the function into all derivative functions
// by applying the Type 2 rules to the function passed in as a parameter
void Transform::negateVar(unsigned int num, int t2Flag) {
   unsigned int newNum;

   // For each item in the Type 2 rule list: swap the bits and pass
   // this new function to the Type 3 transformation method
   for (int i = 0; i < s2; i++) {
      newNum = swapBits(num, type2[i]);
      negateOut(newNum, t2Flag);
   }
}

// Type 3 Transformation
// There are no "rule," per se, for type 3 transformations as
// there is no pattern of bit swapping that will achieve the
// transformation. Instead, the original, and the inverted function
// are considered
void Transform::negateOut(unsigned int num, int t2Flag) {
   // Apply type4 transform to the original passed function
   typeFour(num, t2Flag);

   // Apply type4 transform to the inverted passed function
   num ^= mask;
   typeFour(num, t2Flag);

}
```

```
// Type 4 Transfomation
// This method transforms the function into all derivative functions
// by applying the Type 4 rules to the function passed in as a parameter
void Transform::typeFour(unsigned int num, int t2Flag) {
   unsigned int newNum;

   // For each item in the Type 4 rule list: swap the bits and add
   // the function to the list of derivative functions
   for (int i = 0; i < s4; i++) {
      newNum = swapBits(num, type4[i]);
      addFunction(newNum);
   }
}

// Add the function to the list of derivative functions
void Transform::addFunction(unsigned int num) {
   // Hard coded version uses array instead of something more advanced
   // This is due to speed requirements.
   if (num < numFn) {
      // mark the function as found (change array value to TRUE).
      arr[num] = 1;
   }
}

// Swap the order of the bits of the Integer based on the inputted
// order
unsigned int Transform::swapBits(unsigned int num, int *order) {
   // Hard coded for 3, 4 and 5 variables.
   a[0]  = (num >> vecLen-1) & 1;
   a[1]  = (num >> vecLen-2) & 1;
   a[2]  = (num >> vecLen-3) & 1;
   a[3]  = (num >> vecLen-4) & 1;

   a[4]  = (num >> vecLen-5) & 1;
   a[5]  = (num >> vecLen-6) & 1;
   a[6]  = (num >> vecLen-7) & 1;
   a[7]  = (num >> vecLen-8) & 1;

   if (numVar > 3) {
      a[8]  = (num >> vecLen-9) & 1;
      a[9]  = (num >> vecLen-10) & 1;
      a[10] = (num >> vecLen-11) & 1;
      a[11] = (num >> vecLen-12) & 1;

      a[12] = (num >> vecLen-13) & 1;
      a[13] = (num >> vecLen-14) & 1;
      a[14] = (num >> vecLen-15) & 1;
      a[15] = (num >> vecLen-16) & 1;

      if (numVar > 4) {
         a[16] = (num >> vecLen-17) & 1;
         a[17] = (num >> vecLen-18) & 1;
         a[18] = (num >> vecLen-19) & 1;
         a[19] = (num >> vecLen-20) & 1;

         a[20] = (num >> vecLen-21) & 1;
         a[21] = (num >> vecLen-22) & 1;
         a[22] = (num >> vecLen-23) & 1;
         a[23] = (num >> vecLen-24) & 1;
         a[24] = (num >> vecLen-25) & 1;
         a[25] = (num >> vecLen-26) & 1;
         a[26] = (num >> vecLen-27) & 1;
         a[27] = (num >> vecLen-28) & 1;

         a[28] = (num >> vecLen-29) & 1;
         a[29] = (num >> vecLen-30) & 1;
         a[30] = (num >> vecLen-31) & 1;
         a[31] = (num >> vecLen-32) & 1;
      }
   }

   // -- using the order, add the value stored in a[] to the end of the --
   // function. By the end of the loop, all bits will be represented.
   unsigned int finNum = 0;

   for (int z = 0; z < vecLen; z++) {
      finNum = (finNum << 1) | a[order[z]];
   }
   // -------------------------------------------------------------------

   return finNum;
}
```

C.6 Misc

C.6.1 Misc.h

```
/*
 *  Misc.h
 *  ClassifyCPP
 *
 *  Created by Neil Anderson on 2006-08-11.
 *  Copyright 2006 Neil Anderson. All rights reserved.
 *
 */

#include "main.h"

int getPrecision(int);
int bvtoi(vector<int>);
vector<int> itobv(int, int);

int **allocArr(int, int);
void deallocArr(int **);

string intToString(int);
long double fac(int, int);
long double C(int, int);

bool fileExists(const string&);
```

C.6.2 Misc.cpp

```
/*
 *  Misc.cpp
 *  Classify
 *
 *  A library of useful functions used throughout the code.
 *
 *  Created by Neil Anderson on 2006-08-11.
 *  Copyright 2006 Neil Anderson. All rights reserved.
 *
 */

#include "Misc.h"

// Return the number of bits needed to represent a given integer.
int getPrecision(int x) {
   return ((log (x) / log (2)) + 1);
}

// Convert a vector of binary values into an integer
// Ex:
// [0][1][1][1][0][0][1][1]
//      becomes:
// 01110011 (115 in decimal)
int bvtoi(vector<int> v) {
   int result = 0;
   int s = v.size();

   for (int i = 0; i < s; i++) {
      result = (result << 1) | v[i];
   }
   return result;
}

// Convert a an integer into a vector of binary values
// Ex:
// 01110011 (115 in decimal)
//      becomes:
// [0][1][1][1][0][0][1][1]
vector<int> itobv(int value, int prec) {
   vector<int> result;

   for (int i = 0; i < prec; i++) {
      result.push_back((value >> (prec-i-1)) & 1);
   }

   return result;
}

// Dynamically allocate a 2D array.
int **allocArr(int numRows, int numCol) {
   int **ppi = new int*[numRows];
   int *curPtr = new int[numRows * numCol];

   for (int i = 0; i < numRows; i++) {
      *(ppi + i) = curPtr;
      curPtr += numCol;
   }

   return ppi;
}

// Clean up the allocArr call when finished with the data.
void deallocArr(int ** arr) {
   delete[] *arr;
   delete[] arr;
}
```

```
// Factorial. A cutoff was added so that it can be used
// for binomial coefficients for large numbers. The cutoff
// is used as part of the simplifaction of the binomial
// coefficient equation before computation.
long double fac(int n, int cutoff = 0) {
   long double f = 1.0;

   int i = 1;

   if (cutoff !=0)
      i += cutoff;

   for (; i <= n; i++) {
      f *= (long double)i;
   }

   return f;
}

// Binomial coefficient.
long double C(int n, int r) {
   long double result = 0;

   if (r < 0 || r > n)
      result = 0;
   else {
      // Simplify the equation by using the factorial's
      // curoff function
      if (r > (n - r)) {
         result = (long double)(fac(n, r)/fac(n-r));
      } else {
         result = (long double)(fac(n, (n-r))/fac(r));
      }
   }
   return result;
}

// Convert an int to a string. Used for concatenating ints to strings (like
// adding a loop counter number to the end of a file name)
string intToString(int num) {
   ostringstream myStream;
   myStream << num << flush;

   return(myStream.str());
}

// Check to see if a file already exists.
bool fileExists(const string& fileName) {
   fstream fin;
   fin.open(fileName.c_str(), ios::in);
   if(fin.is_open()) {
      fin.close();
      return true;
   }
   fin.close();
   return false;
}
```