Using GF(2) Matrices in Simulation and Logic Synthesis

Peter M. Maurer, Dept. of Computer Science, Baylor University, Waco, Texas, Peter_Maurer@Baylor.edu

Abstract

GF(2) matrices are matrices of ones and zeros under modulo 2 arithmetic. Like the GF(2) polynomials used in error detection and correction, they have many potential uses in Electronic Design Automation (EDA). Non-singular matrices can be used to define new classes of symmetry called *conjugate symmetries*. Conjugate symmetries have been used to speed up certain kinds of functional-level simulations, and have other potential uses. GF(2) matrices can also be used to transform Boolean vector spaces and simplify Boolean functions. Although matrix transformations can be complex, simpler *single-bit matrices* can be used instead of general matrices. This simplifies the approach without loss of generality. Singular matrices can be used to reduce the complexity of certain functions, beyond what is normally possible with conventional simplification techniques. GF(2) matrices can also be used to define exotic symmetries called s*trange symmetries* and *collapsed symmetries*. These exotic symmetries may prove useful in future EDA applications.

1. Introduction

Symmetry is an inherent property of many Boolean functions that can be exploited in a number of different ways. The most common use is in layout and place-and-route applications where function symmetries create equivalent pins that can be exploited to simplify routing. Circuit comparators that operate at the gate level must take symmetry into account to avoid producing too many false negatives during the comparison process. The problem of detecting and exploiting symmetry has been well studied, some of the more important work can be found in references [1-5].

There are three different types of symmetry that can be exploited, total symmetry, partial symmetry and weak symmetry, which are exemplified by the three gate types NAND, AOI321 and AOI22. (See Figure 1.) NAND exhibits total symmetry because the inputs can be permuted, or rearranged, in an arbitrary fashion. AOI321 exhibits partial symmetry because the first three inputs can be permuted arbitrarily, and the fourth and fifth inputs can be permuted arbitrarily, but inputs cannot be permuted between groups. In this sense, partial symmetry is total symmetry that is restricted to a subset of inputs. The term “weak symmetry” describes a wide class of symmetries that can be both weaker and more powerful than partial symmetry. A function exhibits weak symmetry if its inputs can be permuted in some way that does not correspond to either total or partial symmetry. The AOI22 gate exhibits such symmetry. The first two inputs can be permuted arbitrarily, and the last two inputs can be permuted arbitrarily. It is also possible to

permute the two groups of inputs. Because of its complexity and variety, weak symmetry is difficult to exploit completely in layout and place-and-route. However circuit comparators are generally required to take weak symmetries into account to avoid false negatives.

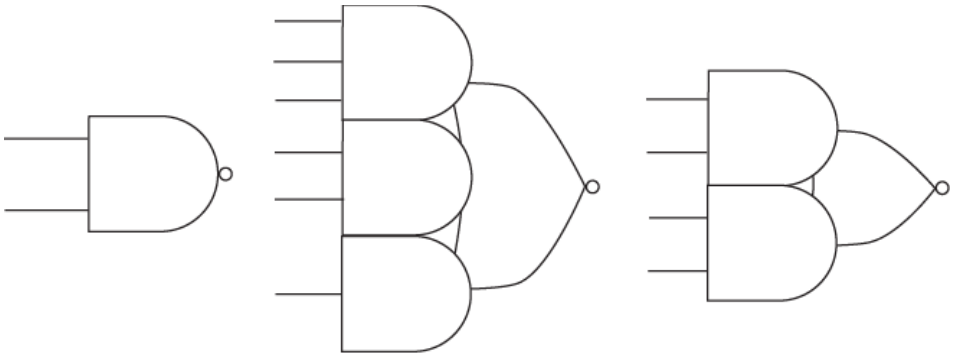

Figure 1. Three Types of Symmetry.

Another important property of Boolean circuits is functional equivalence. Functional equivalences exist when the available signals are not independent of one another. They can be exploited to simplify both the design and layout of digital circuits. The classic example of the exploitation of functional equivalence is mixed-level design, in which the equivalence between a signal and its inverse is used to simplify the design process. Although inversions are the simplest type of functional equivalences to, there are many others. Suppose we have three signals available, A, B, and $A \oplus B$, where $\oplus$ represents the exclusive-or function. It is possible to eliminate any one of these signals because any one of these signals can be computed from the other two. The choice of which signal to eliminate depends on the functions that need to be computed, and $A \oplus B$ is not automatically the best choice.

When functional equivalence is combined with symmetry, new types of symmetries arise. For example the function $f(a,b) = a' + b$ is symmetric with one input inverted with respect to the other. This is a combination of total symmetry and the inversion functional equivalence. This type of symmetry can be exploited in dual rail designs and in applications like the inversion algorithm[6] that are insensitive to inversions. We call this type of symmetry *skew symmetry*. (This term means something slightly different when dealing with real-valued functions.)

Prior work has dealt with functional simulations of logic gates and more complex Boolean functions[7-23]. These include techniques that can take advantage of total, partial and skew symmetries[21-23]. These simulation techniques are significantly more efficient if the functions being simulated are symmetric, but the relative scarcity of totally symmetric functions makes this feature relatively difficult to exploit. Because of this, techniques have appeared that began exploiting more complex types of functional dependencies and the symmetries that arise with respect to them[24]. Continuing investigations along these lines have produced an amazing variety of results, only some of which are directly useful in our simulation. The purpose of this paper is to detail those results that are directly useful in EDA applications, and to give a general view of the state of the art in this area. We are hopeful that by presenting these results we will be opening

up this fascinating area to further research by other members of the design automation community.

2. Linear Transformations.

Given a set of signals, $\{A_1, \cdots, A_n\}$, we can treat these signals as a Boolean vector and perform a linear transformation on them to create a new set of signals. If the linear transformation is non-singular (i.e. one-to-one) then the new set of signals will be functionally equivalent to the original. Any one-to-one function can be used to create a functionally equivalent set of signals, but linear transformations are appealing because they are simple and the underlying theory is well-known. Furthermore, the mathematical theory of symmetry can be couched in terms of linear transformations, making them even more appealing. (For a full mathematical development, see any text on group theory such as [25] or any text on representations of finite groups such as [26].)

First we must begin with a strict mathematical definition of symmetry. This may seem like splitting hairs, but it is precisely this approach that enables us to explore a wide new class of symmetries that are not apparent using a more intuitive approach.

Given a finite set X_n with n elements, a *permutation* on X_n is defined to be a one-to-one function from X_n to itself. It doesn't much matter what X_n is, so we usually assume that $X_n = \{1, 2, 3, \cdots, n\}$, the integers from 1 to *n*. Given a permutation, *p*, and Boolean function, *f*, with *n* inputs, we would like to use *p* to rearrange the inputs of *f* in a way that is obvious and intuitive. However, strictly speaking, *p* can only be used to rearrange the elements of X_n. It cannot be applied to the inputs of a Boolean function. The first step in making the mathematical connection is to treat f as a vector function that is applied to a single input consisting of an *n*-element Boolean vector. Next, we will assume that the *n*-element vectors are elements of an n-dimensional vector space over the field GF(2). GF(2) is simply the integers modulo 2. It contains only the elements 0 and 1. The AND and XOR functions take the place of multiplication and addition. Now, instead of rearranging the variables of an *n*-input function we are rearranging the elements of *n*-dimensional vectors. Since p cannot be applied to vectors any more than it can be applied to function variables, we need to make a connection between *p* and a function T_p that maps vectors to vectors. Fortunately, these sorts of functions are well known. They are the linear transformations that are represented by the *permutation matrices*. A permutation matrix is a matrix that has a single 1 in each row and each column and zeros elsewhere. For every permutation *p* on a set of *n* elements, there is a unique $n \times n$ matrix T_p that corresponds to it.

For example, consider the set of all permutations on a set of 3 elements. This is denoted S_3 and is called the *symmetric group of order* 3. (In general, the set of all permutations on a set of n elements is denoted S_n and is called the *symmetric group of order n*.) S_3 has six elements. (in general, S_n has $n!$ elements.) We can denote these six elements as an arrangement of the numbers 1, 2 and 3. Thus $S_3 = \{123, 132, 213, 321, 231, 312\}$. The matrices corresponding to these six permutations are are given in Figure 2.

$$\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}\begin{pmatrix}1&0&0\\0&0&1\\0&1&0\end{pmatrix}\begin{pmatrix}0&1&0\\1&0&0\\0&0&1\end{pmatrix}\begin{pmatrix}0&0&1\\0&1&0\\1&0&0\end{pmatrix}\begin{pmatrix}0&1&0\\0&0&1\\1&0&0\end{pmatrix}\begin{pmatrix}0&0&1\\1&0&0\\0&1&0\end{pmatrix}$$

Figure 2. The Standard Representation of S_3.

These matrices are all non-singular, and taken together form a group under matrix multiplication. The set of all non-singular $n \times n$ matrices over the field GF(2) is called the *general linear group of order n over GF(2)*, and is denoted $GL_n(2)$. The set of all $n \times n$ permutation matrices is called the *standard representation* of S_n in $GL_n(2)$, and is denoted $SR_n(2)$.

An *n*-input Boolean function f is said to be totally symmetric if for each $M \in SR_n(2)$, $fM(v) = f(M(v)) = f(v)$ for all v. That is to say, the composition of f and M, is identical to f. This definition corresponds to our usual intuitive understanding of symmetric functions.

The key point here is that the standard representation of S_n is not the only representation of S_n in $GL_n(2)$. There are many others, one of which is given in Figure 3.

$$\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}\begin{pmatrix}1&0&0\\1&1&1\\0&0&1\end{pmatrix}\begin{pmatrix}1&0&0\\1&0&1\\1&1&0\end{pmatrix}\begin{pmatrix}1&0&0\\0&1&0\\0&1&1\end{pmatrix}\begin{pmatrix}1&0&0\\1&1&1\\1&1&0\end{pmatrix}\begin{pmatrix}1&0&0\\1&0&1\\0&1&1\end{pmatrix}$$

Figure 3. An Alternative Representation of S_3.

Each of these representations defines an entirely new class of symmetry, with total symmetries, partial symmetries and weak symmetries. The symmetries with which we are familiar are only those generated by the standard representation.

The simplest way to generate a new representation of S_n is to start with the standard representation, and compute the conjugate of $SR_n(2)$ with respect to a non-singular linear transformation *T*. If M is any matrix and T is a non-singular matrix with inverse T^{-1}, the matrix $T^{-1}MT$ is called *the conjugate of M with respect to T*. In some cases $M = T^{-1}MT$, but in general the two matrices are different. (Conjugate matrices are also called *similar* matrices.) To find the conjugate of $SR_n(2)$ with respect to T, one computes the conjugate of each matrix in $SR_n(2)$ with respect to T. The two representations given in Figure 2 and Figure 3 are $SR_3(2)$, and the conjugate of $SR_3(2)$ with respect to the matrix of Figure 4.

$$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 4. A Symmetry Matrix.

The symmetries generated by representations conjugate to $SR_n(2)$ are called *conjugate symmetries*. In cases where the matrix T is not obvious we will speak of *conjugate symmetry with respect to T*. Within conjugate symmetries we have total symmetries, partial symmetries, weak symmetries and skew symmetries.

There is a natural relationship between conjugate symmetries and symmetric Boolean functions. Let R be a representation of S_n in $GL_n(2)$, an *n*-input Boolean function f is said to be symmetric with respect to R, if for every matrix $M \in R$, $fM(v) = f(M(v)) = f(v)$ for all vectors v. Now, let R be a conjugate of $SR_n(2)$ with respect to the non-singular matrix T. If f is an *n*-input totally symmetric Boolean function, then the function fT is symmetric with respect to R. It is easy to show that this is the case, for $fTT^{-1}MT(v) = fMT(v) = fT(v)$.

Our interest in conjugate symmetries originally stemmed from the fact that one of our most powerful simulation algorithms [23,27] is significantly more efficient if the function being simulated is symmetric. The simulation is based on state machines that are significantly smaller and more efficient than those for symmetric functions. If the Boolean function, g, can be factored into $g = fT$ where f is symmetric, and the cost of the transformation T is negligible, then we have a significant opportunity to improve the performance of our simulations.

In our simulation engine it is easy to implement T at negligible cost. A function simulation consists of three types of state machines, input state machines, gate state machines, and output state machines. The input state machines are used to process the inputs to a function. Typically they toggle between two states which may or may not correspond to the 1/0 state of the physical input.

The gate state machine is used to determine when event propagation occurs. The gate state machine may represent a conventional gate or a more complex function. It processes signals from several input state machines and signals the output state machine when an output transition occurs.

The output state machine is used to schedule events or to cancel previously scheduled events. It is used to schedule the input state machines for the functions in the fanout of the current function. Since two successive changes in a net cancel one another, the output machine toggles between the *scheduled* and *unscheduled* state. If the final state of the machine is the *scheduled* state, it will be forced back to the unscheduled state by the input state machines of the following functions.

In a conventional simulation the inputs of the function and the input state machines are in one-to-one correspondence. When an event occurs on a function input, an event is directed to a single input state machine. For simulations using conjugate symmetry, this correspondence is broken. An event on a particular input may be directed to more than one state machine. The input state machines function in such a way as to compute the exclusive or of a collection of events. The events directed to a particular machine are

determined by the conjugacy matrix, T. For example, Figure 5 shows a 3-input Boolean function that is symmetric with respect to the matrix of Figure 4.

a,b,c	Output
0,0,0	1
0,0,1	0
0,1,0	1
0,1,1	0
1,0,0	1
1,0,1	0
1,1,0	1
1,1,1	0

Figure 5. A Function With Conjugate Symmetry.

The function of Figure 5 can be factored onto fT, where $f = a'b'c' + ab'c + a'bc + abc'$. Note that f is totally symmetric.

The columns of the matrix T determine how events are routed to the input state machines. Each column represents an input state machine, and each row represents a function variable. The simulation of f will have three input machines which we will number 1, 2 and 3. Events from variable a will be routed to machine 1, events from variables a and b will be routed to input machine 2, and events from variables c and d will be routed to input machine 3. Figure 6 shows how events are routed in an ordinary 3-input machine, and how they are routed using the conjugacy matrix of Figure 4. This scheme causes extra events to be generated for each input-change, but the cost of processing these events is negligible.

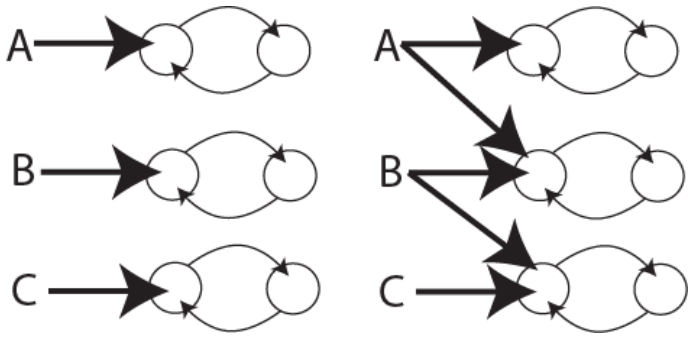

Figure 6. Conjugate Symmetry Routing.

The main problem is factorization. That is, given g, find f and T such that $g = fT$. Our factorization technique is based on our technique for detecting symmetries, which uses *n*-dimensional lattices. We start with an *n*-dimensional hypercube whose vertices are labeled with the values of the function. By comparing the function values along various diagonals we are able to detect partial symmetries with respect to pairs of variables. Figure 7 illustrates the detection of ordinary symmetry and skew symmetry for a two-input function. The vertices of the hypercubes are labeled with the input values. The function must have the same value for the two circled inputs for the corresponding symmetry to exist.

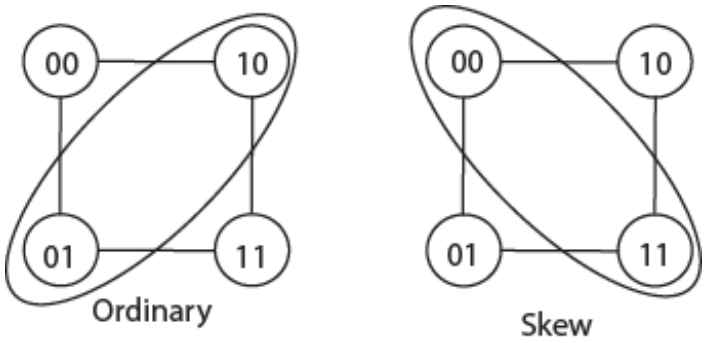

Figure 7. Symmetry Detection.

By combining variables into "hypervariables" we are able to continue the process until all partial and total symmetries have been detected. Combining two variables produces non-cubic, hyperlinear structures such as that pictured in Figure 8. Like the original hyper-cube structure, a hyperlinear structure can be viewed as a lattice. (We use functional-level comparisons to avoid generating the entire hypercube or hyperlinear structure, but the net effect is the same.) Our technique detects skew symmetries by observing that a skew symmetry will cause the hypercube or hyperlinear structure to be inverted in one dimension. By using the opposite set of diagonals for our comparison, we can adjust for the reversal.

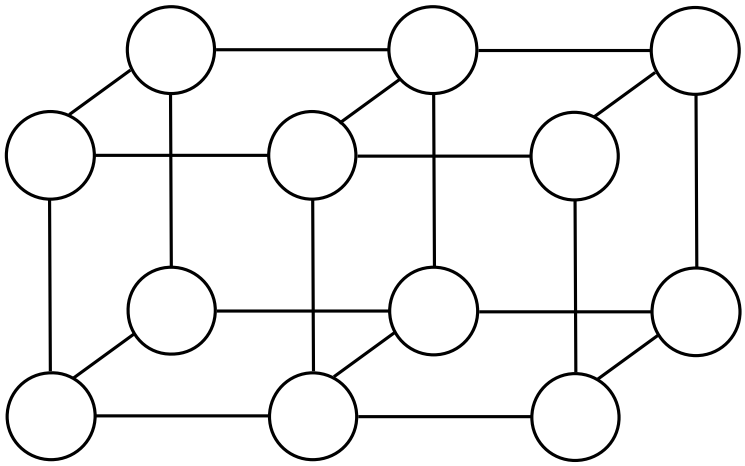

Figure 8. A Hyperlinear Structure.

Our symmetry detection algorithm can be adapted to detect conjugate symmetries by observing the effect of a conjugacy matrix on the function lattice. Consider the matrix of Figure 9, which has a main diagonal of all 1's, a single 1 off the main diagonal, and zeros elsewhere. (This is an example of a *single-bit matrix* which we will discuss in depth later.)

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 9. A Single-Bit Matrix.

Suppose f is a four-input function with inputs *a*, *b*, *c*, and *d*, and that f is symmetric with respect to the matrix of Figure 9. This matrix represents the variable *c* being *conditionally inverted* by the variable *a*. A conditional inversion reverses certain sub-planes of the hypercube or hyperlinear structure, and leaves others intact. Assuming that we number the planes starting with zero, the odd numbered planes will be inverted and the even numbered planes will be left intact. To check for a conjugate symmetry with respect to a single-bit matrix, simply indexing the odd numbered planes in reverse order will compensate for the conditional inversion. When checking for symmetries with respect to two variables, *a*, and *b*, say, it is necessary to do two separate checks, because *a* can be conditionally inverted by *b*, and *b* can be conditionally inverted by *a*.

When a symmetry with a conditional inversion is detected, we reverse the affected planes of the hyperlinear structure, record the single-bit matrix, and collapse the hyperlinear structure just as if the symmetry had been ordinary. We are able to detect total, partial and skew conjugate symmetries.

Once all symmetries have been found, we multiply all single-bit matrices together in the order they were found. The resultant hyperlinear structure is used as the gate state machine of the function *f* and the product matrix is the symmetry matrix T. After creating the input state machines and routing the input events according to the columns of the symmetry matrix, T, the resultant simulation structure will simulate the original function $g = fT$.

For example, consider the function $f(a,b,c,d) = ac' + bc' + bd' + ab'd + a'cd'$. This function has no ordinary or skew symmetries. This example also shows how we used functional-level comparisons to avoid generating the complete hypercube. First we eliminate variable a by setting $a = 0$ and then setting $a = 1$. The two resultant expressions are placed in a one-dimensional hypercube as shown in Figure 10.

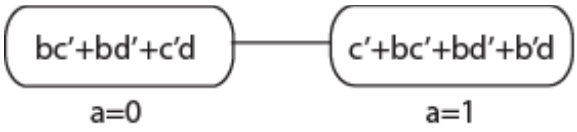

Figure 10. Eliminating a.

Next we eliminate the variable b in the same way and create the two dimensional hypercube of Figure 11.

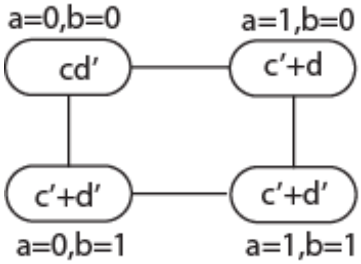

Figure 11. Eliminating a and b.

If we reverse the right column of Figure 11, the diagram will contain an ordinary symmetry. This implies that a and b are symmetric with a conditionally inverting b. The matrix T_1 of Figure 12 defines this relationship.

$$T_1 = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 12. The First Symmetry Matrix.

Next, we swap the two equations in the right hand column, and combine the two states containing $c' + d'$ giving the hyperlinear structure of Figure 13. We also save the matrix T_1 for future use.

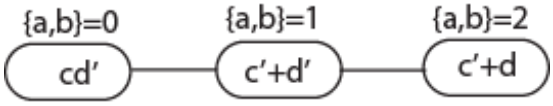

Figure 13. Collapsing the State Machine 1.

Next, we eliminate the variable c, giving the structure of Figure 14.

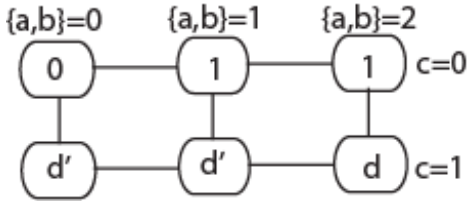

Figure 14. Eliminating c.

This diagram will exhibit an ordinary symmetry if we reverse the center column, which corresponds to the combined variable $\{a,b\}$ being conditionally inverted by c. The new hyperlinear structure is shown in Figure 15.

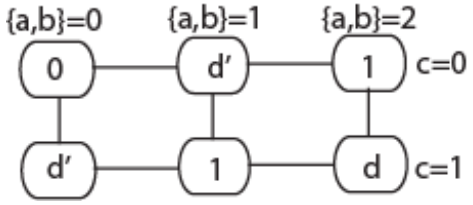

Figure 15. The Second Transformation.

Because $\{a,b\}$ is not a simple variable, we cannot use a single-bit matrix to represent the conditional inversion. Instead we must use matrix T_2 of Figure 16.

$$T_2 = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 16. The Second Symmetry Matrix.

Again, we save the matrix T_2 and combine the states containing 1 and the states containing d' to obtain the hyperlinear structure of Figure 17.

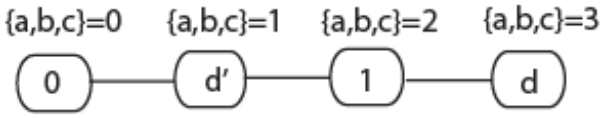

Figure 17. The Second Collapse.

Eliminating the final variable d gives us the structure of Figure 18.

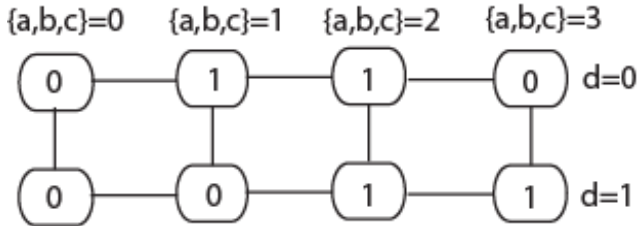

Figure 18. Eliminating d.

If we reverse the second and fourth columns, the diagram of Figure 18 will contain an ordinary symmetry, which implies that the combined variable $\{a,b,c\}$ is conditionally inverted by d. This condition is generated by the matrix T_3 of Figure 19.

$$T_3 = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 19. The Final Symmetry Matrix.

The matrix T_3 is saved along with the others and the columns of the state diagram are reversed to produce the diagram of Figure 20.

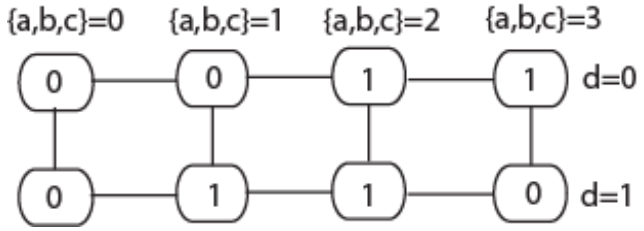

Figure 20. The Third Transformation.

Finally, we combine the diagonals containing 1's and the diagonal containing 0's to produce the final state machine of Figure 21.

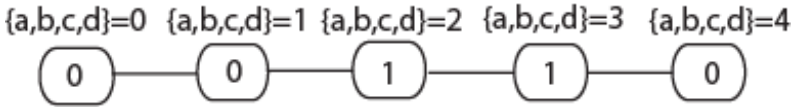

Figure 21. The Final State Machine.

The final symmetry matrix is computed from T_1, T_2, and T_3 as shown in Figure 22.

$$T_1 \times T_2 \times T_3 = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 22. The Final Symmetry Matrix.

Note that although T_2 and T_3 are not single-bit matrices, they are products of single-bit matrices as shown in Figure 23. Thus conditionally inverting a combined variable is equivalent to conditionally inverting the individual variables.

$$T_2 = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$T_3 = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 23. Single-bit Breakdowns.

3. Single-bit Matrices

The previous section made extensive use of single-bit matrices. This approach may seem restrictive but this section will show that any non-singular matrix can be represented as a product of single-bit matrices. Thus, we have not restricted ourselves by taking this approach. It turns out that single-bit matrices are useful in other algorithms as well, so the results of this section are widely applicable.

First, let us begin by providing a formal definition of single-bit matrices. This level of formality is necessary to make the proofs of the theorems rigorous.

Definition. An $n \times n$ matrix M is called a *single-bit matrix* if $a_{i,i} = 1$ for all i, $1 \le i \le n$, and there exists a single pair of integers $0 \le i, j \le n$, with $i \ne j$ such that $a_{i,j} = 1$. For all other pairs of indices $1 \le k, m \le n$, $a_{k,m} = 0$. We designate the single-bit matrix with a 1 in row i and column j as $S_{i,j}$.

The theorems in this section will give a step by step procedure for factoring any non-singular matrix into a product of single-bit matrices. Given two non-singular matrices T, and S, one can factor S out of T by finding the inverse, S^{-1}, of S, and multiplying T by S^{-1}. In other words, if $T = S \times Q$, then $Q = S^{-1} \times T$. Our first theorem shows that every single-bit matrix is its own inverse so if $T = S_{i,j} \times Q$, then $Q = S_{i,j} \times T$. That is, we can factor a single-bit matrix $S_{i,j}$ out of T by multiplying T times $S_{i,j}$.

Theorem 1. Any single-bit matrix is of order 2. That is, if A is a single bit matrix, then A^2 is the identity matrix.

Corollary 1: For any matrix T, $S_{i,j} \times (S_{i,j} \times T) = T$.

Given a non-singular matrix, T, we know from linear algebra that it is possible to perform a Gaussian elimination on T to convert it to the identity matrix. We will show that we can perform each step of the Gaussian elimination by multiplying T by the appropriate single-bit matrices. Although the theorems are stated in somewhat stilted

form for the sake of rigor, the steps in the proof will seem more straightforward if one keeps in mind that when we pre-multiply a matrix T by a single-bit matrix $S_{i,j}$ the result is a matrix Q in which every row except row i is identical to the corresponding row of T, and row i of Q will be the sum of rows i and j of T.

The first step in the Gaussian Elimination algorithm is to create an upper triangular matrix in which every element below the main diagonal is zero. This involves two sub-procedures: pivoting to place non-zero elements on the main diagonal, and elimination of non-zero elements below the main diagonal. Theorem 2 shows how to perform the pivot operation if it is required.

Theorem 2. Suppose T is a non-singular $n \times n$ matrix with the first $i-1$ columns in upper triangular form, and that element i,i is zero. Then there exists a j and a single-bit matrix $S_{i,j}$ such that the first $i-1$ columns $S_{i,j} \times T$ are identical to those of T, and element i, j of $S_{i,j} \times T$ is equal to 1.

Corollary 2: If M is a matrix meeting the conditions of Theorem 2, then M can be factored into a pair of matrices $M = N \times P$, where N is a single-bit matrix, and element $a_{i,i}$ of P is equal to 1.

Next we will show that we can factor any non-singular matrix M into a product of single bit matrices and an upper triangular matrix T. To do this, we must eliminate the 1's below the main diagonal. Theorem 3 shows how to do this.

Theorem 3. Let M be a non-singular $n \times n$ matrix with the first $i-1$ columns in upper triangular form ($0 < i < n-1$), and with element i,i equal to 1. Let j be the smallest integer such that $j > i$ and element j,i is equal to 1. Then the first $i-1$ columns of $S_{j,i} \times M$ are identical to the first $i-1$ columns of M and column i of $S_{j,i} \times M$ is identical to column i of M except for element j,i, which is equal to 1 in M and equal to zero in $S_{j,i} \times M$.

Corollary 3. Any $n \times n$ non-singular matrix M can be factored into a set of matrices $M = A_1 \times \cdots \times A_k \times B$ such that A_i, $1 \le i \le k$ is a single-bit matrix, and B is an upper triangular matrix.

Because we are dealing with matrices over GF(2), the upper triangular matrix of Corollary 3 must have 1's on the main diagonal. Theorem 4 shows how to do back-substitution to eliminate any 1's above the main diagonal. Once this procedure is complete, the result will be the identity matrix.

Theorem 4. Let M be an upper triangular matrix. Let j be the largest integer such that column j of M has a 1 above the main diagonal, and let i be the largest integer such that $i < j$ and position i, j of M is equal to 1. Then the product matrix $S_{i,j} \times M$ is

an upper triangular matrix that is identical to M except for position i, j which is equal to one in M and zero in $S_{i,j} \times M$.

Corollary 4. Any $n \times n$ upper triangular matrix M can be factored into a sequence of matrices $M = A_1 \times \cdots \times A_k$, such that A_i, $1 \le i \le n$, is a single-bit matrix.

Theorem 5 is a summary of the results of the preceding theorems.

Theorem 5. Any $n \times n$ non-singular matrix M can be factored into a sequence of single-bit matrices: $M = A_1 \times \cdots \times A_k$.

Even though our algorithms generate matrices by constructing a product of single-bit matrices, Theorem 5 shows that these algorithms can still be general purpose.

4. Products of Single-Bit Matrices.

The decomposition given in the preceding section can produce a sequence of matrices with many duplicates. This series of matrices can be simplified by combining duplicate matrices, but to do this one must be aware of which single-bit matrices commute with one another. For example, the matrices $S_{1,2}$ and $S_{3,4}$ commute with one another, but $S_{1,2}$ and $S_{2,1}$ do not. The purpose of this section is to characterize all products of two single-bit matrices and show which matrices commute with one another. This will facilitate reducing the length of a decomposition sequence. Since any single-bit matrix is its own inverse, we can confine ourselves to sequences of matrices $\cdots \times S_{i,j} \times S_{m,n} \times \cdots$ where $S_{i,j} \neq S_{m,n}$. We should also note that since $S_{i,j}$ and $S_{m,n}$ are single-bit matrices, it is understood that $i \neq j$ and $m \neq n$. In the remainder of this section we will assume we are dealing with $r \times r$ matrices, that the product in question is $S_{i,j} \times S_{m,n}$, that $a_{p,q}$ will denote elements of $S_{i,j}$, $b_{p,q}$ will denote elements of $S_{m,n}$, and $c_{p,q}$ will denote elements of $S_{i,j} \times S_{m,n}$. For brevity, the proofs of the lemmas and theorems have been omitted from this section. The interested reader will find them in Appendix A. Lemmas 6 and 7 state that the product $S_{i,j} \times S_{m,n}$ is identical to $S_{m,n}$ except for row i which is the sum of rows i and j of $S_{m,n}$.

Lemma 6. If $p \neq i$ then $c_{p,q} = b_{p,q}$.

Lemma 7. If $p = i$ then $c_{p,q} = c_{i,q} = b_{i,q} + b_{j,q}$.

Theorem 8 states that the "extra" one bits of $S_{i,j}$ and $S_{m,n}$ are copied into the product regardless of their relative positions.

Theorem 8. In $S_{i,j} \times S_{m,n}$, $c_{i,j} = 1$ and $c_{m,n} = 1$.

Theorem 9 gives conditions under which nothing happens except copying in the extra one bits.

Theorem 9. If $j \neq m$ then the main diagonal of $S_{i,j} \times S_{m,n}$ is all 1's, and the only non-zero elements off the main diagonal are the elements $c_{i,j}$ and $c_{m,n}$.

Theorem 10 shows one thing that will happen when the conditions of Theorem 9 are not met. The condition of Theorem 10 states that the two extra one bits are diametrically opposed to one another across the main diagonal. When this happens, a zero is introduced into the main diagonal.

Theorem 10. If $j = m$ and $i = n$, then in $S_{i,j} \times S_{m,n}$, $c_{i,i} = 0$ and the only non-zero element of row i is $c_{i,j}$.

Theorem 11 completes the discussion about what happens when the conditions of Theorem 9 are not met. If the two positions "interfere" with one another, but are not diametrically opposed, then an extra one is introduced.

Theorem 11. If $j = m$ and $i \neq n$, then in $S_{i,j} \times S_{m,n}$, $c_{i,i} = 1$, $c_{i,j} = 1$ and $c_{i,n} = 1$. All other elements of row i are equal to zero.

Theorem 12 states that two single-bit matrices commute if and only if Theorem 9 applies to *both* $S_{i,j} \times S_{m,n}$ and $S_{m,n} \times S_{i,j}$.

Theorem 12. The two single-bit matrices $S_{i,j}$ and $S_{m,n}$ commute with one another if and only if $j \neq m$ and $i \neq n$.

5. Simplifying Boolean Functions

Conjugate symmetry imposes a set of functional equivalences on the inputs to a function. If f is a totally symmetric function, and T is a linear transformation, then fT is totally symmetric with respect to a set of linear expressions involving the original inputs. By taking advantage of these functional dependencies, it is possible to significantly simplify the implementation of many Boolean functions. In this case, the linear transformation cannot be obtained for free, so in is usually necessary to use the same linear transformation to simplify several functions.

As an example, consider the set of totally symmetric 4-input Boolean functions. Because the value of a totally symmetric function depends only on the number of 1's in the input, there are 32 such functions. We can designate these functions using a 5-bit binary number such as 10101. The function that corresponds to this "spectrum" would map input 0000 to 1, inputs 0001, 0010, 0100, and 1000 to 0, inputs 0011, 0101, 1001, 0110, 1010, and 1100 to 1, inputs 0111, 1101, 1011, and 1110 to 0, and input 1111 to 1. (The low-order bit corresponds to input 0000.). Table 1 of Appendix B gives a minimal

sum-of-products implementation of these functions. There are a total of 164 implicants, 8 with one factor, 28 with two factors, 56 with 3 factors and 72 with 4 factors. After applying the linear transformation of Figure 24. Matrix for Symmetric Functions.to these functions, we obtain the corresponding minimal forms of Table 2 of Appendix B. In the second table, there are a total of 90 implicants, 17 with one factor, 33 with two factors, 32 with 3 factors and 8 with four factors. Applying the linear transformation not only cuts the number of implicants nearly in half, but also significantly reduces the number of terms.

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 24. Matrix for Symmetric Functions.

When dealing with specific functions, of course, it is necessary to use a more systematic method for choosing an appropriate linear transformation. As in the simulation algorithms, it is easiest to deal with single-bit matrices. Single-bit matrices can be chosen one at a time to perform local optimizations. Once all local optimizations have been performed, the individual matrices can be multiplied together in the order they were originally chosen to perform the global optimization of the entire function.

For example, consider the symmetric function that assigns 1 to the inputs 0111, 1011, 1101, and 1110, and zero to everything else. This function has the Karnaugh map shown in Figure 25. (We assume that inputs 1 and 2 are on the left, and inputs 3 and 4 are on the top.)

	00	01	11	10
00	0	0	0	0
01	0	0	1	0
11	0	1	0	1
10	0	0	1	0

Figure 25. Initial Karnaugh Map.

When applying a single-bit matrix $S_{i,j}$ it is important to remember that $S_{i,j}$ causes input number i to conditionally invert input j. Thus $S_{1,2}$ will cause input 1 to conditionally invert input 2. Conditional inversions take place only when the inverting input is equal to 1, and this corresponds to the last two rows of the Karnaugh map. Applying the transformation swaps the last two rows, giving the map of Figure 26.

	00	01	11	10
00	0	0	0	0
01	0	0	1	0
11	0	0	1	0
10	0	1	0	1

Figure 26. First Map Transformation.

This transformation reduces the essential implicants from four to three. Applying the transformation $S_{3,4}$ swaps the last two columns, giving the map of Figure 27.

	00	01	11	10
00	0	0	0	0
01	0	0	0	1
11	0	0	0	1
10	0	1	1	0

Figure 27. Second Map Transformation.

The combined transformation is shown in Figure 28.

$$S_{1,2} \times S_{3,4} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 28. The Combined Transformation.

The result is to reduce the number of implicants from four to two, and reduce the number of factors in each implicants from four to three. It is not possible to reduce the number of implicants any further using a non-singular transformation, since the four original minterms are linearly independent, and any four minterms that combine into a single implicant must be linearly dependent. However, this problem is addressed, further in Section 6.

To select appropriate single-bit matrices it is necessary to have a clear idea of how each matrix will affect the Karnaugh map. Each row of the matrix affects a particular area of the Karnaugh map and each column transforms the area in a certain way.. Figure 29 shows the areas of the map that correspond to each row of the matrix.

Row 1

	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

Row 3

	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

Row 2

	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

Row 4

	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

Figure 29. Affected Map Areas.

To show how each column transforms the affected area we use variable 1 as an illustration, The other areas are affected in much the same way. In Figure 30, each lettered cell is exchanged with the cell containing the same letter.

Column 2

	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	**A**	**B**	**C**	**D**
10	**A**	**B**	**C**	**D**

Column 3

	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	**A**	**C**	**C**	**A**
10	**B**	**D**	**D**	**B**

Column 4

	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	**A**	**A**	**C**	**C**
10	**B**	**B**	**D**	**D**

Figure 30. Area Transformations.

Linear transformations can also be used with to improve the Quine-McCluskey minimization algorithm. For example, consider the symmetric function given by the minterms, 0011, 0101, 1001, 0110, 1010, 1100, and 1111. The initial table given in Figure 31.

0	0000
1	
2	0011, 0101, 1001, 0110, 1010, 1100
3	
4	1111

Figure 31. The Initial Q-M Table.

In Figure 31, the minimization prospects look slim because there are no adjacent minterms. However, we can use linear transformations to change this. First, we count the number of 1’s in each of the four positions, and then choose the position with the highest

number of 1's and choose that position as the inverting variable. The minterms with 1's in the inverting position are the ones that will move when applying the transformation. The direction of movement depends on the value of the variable being inverted. Inverting a 1 moves the minterm up the table, while inverting a zero moves it down. Because this function is symmetric, it doesn't matter which variable we choose as the inverting variable, so we might as well choose position 1. By the same token, it doesn't matter in this example which variable we choose as the inverted variable, so we might as well choose position 2. This will move four minterms, two up the table, and two down. Of course, we wish to choose the inverted variable so as to maximize the number of neighboring minterms. The result of applying $S_{1,2}$ gives us the table of Figure 32.

0	0000
1	1000
2	0011 0101 0110
3	1101 1011 1110
4	

Figure 32. The First Q-M Transformation.

And now we can proceed with the second column of the table as shown in Figure 33.

0	0000	-000
1	1000	
2	0011 0101 0110	-011 -101 -110
3	1101 1011 1110	
4		

Figure 33. Adding the Second Column.

We have used all minterms, so there are no essential prime implicants among them. The new table has no adjacent implicants, so further minimization is impossible. To deal with this problem we need to find a variable position with no dashes, or with a minimal number of dashes. If a position containing a dash is chosen for the inverting variable, any implicants containing a dash in the inverting position must be broken into two essential implicants of larger size, thus undoing the operation that caused the dash to appear. Clearly it would be best to avoid this, so in the table above, we choose position 2 as the inverting position because it contains no dashes and has a maximal number of 1.s We choose position 3 as the inverted position because this will move one implicant up the table, and one down the table. It is OK to have a dash in an inverted position because dashes are invariant under conditional inversion. The result of applying $S_{2,3}$ gives us the table of Figure 34.

0	0000	-000
1	1000	-100
2	0011 0101 0110	-011
3	1101 1011 1110	-111
4		

Figure 34. The Second Q-M Transformation.

Since there are no essential implicants in column 1 of Figure 34, we do not apply the transformation to that column.

We can now compute column 3, as shown in Figure 35. We have used all implicants of column 2 so there are no essential prime implicants in column 2.

0	0000	-000	--00
1	1000	-100	
2	0011 0101 0110	-011	--11
3	1101 1011 1110	-111	
4			

Figure 35. Adding the Third Column.

We can complete the operation by applying $S_{3,4}$, giving the table of Figure 36. Again, since there are no essential prime implicants in columns 1 and 2, we apply the transformation only to column 3.

0	0000	-000	--00
1	1000	-100	--10
2	0011 0101 0110	-011	
3	1101 1011 1110	-111	
4			

Figure 36. The Third Q-M Transformation.

We can now compute the final column, column 4 as shown in Figure 37.

0	0000	-000	--00	---0
1	1000	-100	--10	
2	0011 0101 0110	-011		
3	1101 1011 1110	-111		
4				

Figure 37. Adding the Final Column.

The final transformation matrix is shown in Figure 38.

$$S_{1,2} \times S_{2,3} \times S_{3,4} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 38. The Final Matrix.

When applying linear transformations, it is important to note that the zeros of a function can be just as important as the ones even when a sum-of-products result is desired. For example, consider the symmetric function given by the minterms 0000, 0001, 0010, 0100, 1000, 0011, 0101, 1001, 0110, 1010, 1100, and 1111. The Quine-McClusky minimization of this function produces the essential prime implicants 00--, 0-0-, -00-, 0—0, -0-0, --00 and 1111. Each of these prime implicants must be included in the minimal cover. Looking at the 1's of this function, there are no apparent opportunities for further minimization through application of linear transformations. However, the zeros of this function are 0111, 1011, 1101, and 1110. These zeros can be clustered using the linear transformation $S_{1,2} \times S_{3,4}$, producing the minimal cover 00--, --00, -1-1, -010, -0-0, reducing the number of terms from 7 to 3, and eliminating the 4-factor implicant 1111. The transformation $S_{1,2} \times S_{2,3} \times S_{3,4}$ does an even better job yielding the minimal cover ---0, 00--, -11-.

6. Collapsing Boolean Functions

In the preceding section, we have seen how replacing the inputs of a function with a linear combination of those inputs can significantly reduce the complexity of a function. Despite this simplification, however, the number of inputs to the function does not change. In this section we will show how to use linear transformations to reduce the number of inputs to a function. In this case we will be using a singular linear transformation, N. Given an n input function f we seek a singular linear transformation N and a function g such that $f = gN$. Since N is singular, the dimension of the range space of N (called the *rank* of N) is smaller than n. This does not necessarily imply that g is a function of fewer than n inputs, but it is easy to show that if N and g exist then we can always find a g' and N' such that $f = g'N'$, and g' has fewer than n inputs.

For example consider the 4-input function f with minterms {0100,0010,0011,1011,1100,1101}, and the singular 4×4 matrix shown in Figure 39.

$$N = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}.$$

Figure 39. A Singular Transformation.

For a function g to exist such that $f = gN$, the following must be true. For every pair of vectors v_1, v_2 such that $v_1 \neq v_2$ and $N(v_1) = N(v_2)$, $f(v_1)$ must equal $f(v_2)$. To verify that this is indeed the case, we construct the table of Figure 40.

v	$N(v)$	$f(v)$
0000	0000	0
0001	1001	0
0010	0011	1
0011	1010	1
0100	0110	1
0101	1111	0
0110	0101	0
0111	1100	0
1000	1100	0
1001	0101	0
1010	1111	0
1011	0110	1
1100	1010	1
1101	0011	1
1110	1001	0
1111	0000	0

Figure 40. Collapsing the Function *f*.

A quick inspection of Figure 40 will verify that the required condition is satisfied, and that g has the minterms {0011,1010,0110}. Although g is simpler than f it is still a function of four inputs. Now consider the singular 4×4 matrix N' of Figure 41.

$$N' = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Figure 41. A Second Singular Transform.

The matrix N' can also be used to simplify f as the table of Figure 42 shows.

v	$N'(v)$	$f(v)$
0000	0000	0
0001	0111	0
0010	0010	1
0011	1100	1
0100	0100	1
0101	1010	0
0110	0110	0
0111	1000	0
1000	1000	0
1001	0101	0
1010	1010	0
1011	0100	1
1100	1100	1
1101	0010	1
1110	1110	0
1111	0000	0

Figure 42. Collapsing Function *f* Again.

Thus N' will work, and g' has the minterms {0010,1100,0100}, making it a function of three variables.

Surprisingly enough, the compatibility of f with N and N' is dependent only on f as Theorem 13 shows. (Again the proof is in Appendix A.)

Theorem 13. Let f be a function of n variables, and N be a singular $n \times n$ matrix such that any subset of size $n-1$ or smaller of the rows of N is linearly independent, but the sum of all n rows is the zero vector. Then N is of rank $n-1$ and f is compatible with N if and only if $f(v) = f(\overline{v})$ for all n element vectors v, where $\overline{v}$ is the bit-wise complement of v.

The compatibility with a singular transformation N does not guarantee compatibility with all singular transformations even if they are of the same rank. The following four transformations are all of rank 3, but compatibility with one does not guarantee compatibility with any of the others. The rows of transformations A, B, and C do not sum to zero, so they do not meet the conditions of Theorem 13.

$$N = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix} \quad A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \quad C = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 43. Singular Transformations of Rank 3.

Matrix C is an interesting case because compatibility with Matrix C and its conjugates is the basis of the traditional function simplification techniques such as Karnaugh maps and the Quine-McCluskey algorithm.

A complete discussion of singular linear transformations and their compatible functions is extremely complex and beyond the scope of this paper. However, the rule $f(v) = f(\overline{v})$ is an easy test that can be applied to find functions that can be simplified using this method.

7. Strange and Collapsed Symmetries

Strange and collapsed symmetries extend the concept of symmetry beyond the concepts of conjugate symmetry. Although the area is intriguing and full of potential, there are many, many problems that still remain to be solved. This is partially due to the fact that there are a few unsolved mathematical problems in the area. We will present some of our major results in this area with the hope that the ideas presented here may stimulate further research.

7.1. Strange Symmetries.

As noted above, Boolean function symmetries are defined by a set of linear transformations. In essence, we start with the group S_n, and map it into a group of $n \times n$ matrices. We have already explored the standard representation, which gives us total, partial, weak, and skew symmetries. We have also noted that each conjugate of the standard representation defines a whole new class of symmetries containing its own total, partial, weak and skew symmetries.

However, there are many representations of S_n that are not conjugate to the standard representation. We call the symmetries generated by these representations *strange symmetries*, because they are difficult to characterize with existing terminology. It is not yet clear that there is any advantage to a function being strangely symmetric, but these classes appear to contain functions that can be factored in interesting ways. Strange symmetries do not exist for 2 and 3 input functions, but appear to exist for functions with 4 or more inputs. (We have studied only 4 and 5 input functions.) For 4-input functions, there are nine super-classes of representations, one of which contains the standard representation and its conjugates, and the other eight of which are strange. There appear to be 4 such super-classes for 5 input functions, but the large number of non-singular 5×5 matrices, and the relatively large size of S_5 makes these results difficult to verify.

Since the strange super-classes do not contain permutation matrices, it is difficult to identify a particular representative of each to call “The Standard Representation.” Moreover, the groups themselves provide little insight into the nature of the super-classes. Thus, is more meaningful to analyze the *orbits* of each class.

Suppose G is a group of $n \times n$ matrices. The n element vectors v_1 and v_2 are in the same orbit of G if and only if there is a matrix $M \in G$ such that $M(v_1) = v_2$. Being in the same orbit is an equivalence relation, and the *structure* of the orbits of a group is invariant under conjugation. That is, the number and size of the orbits will be the same for G and $T^{-1}GT$.

The orbits of the standard representation of S_4 are defined by the number of ones in each vector. Two vectors are in the same orbit if and only if they have the same number of ones. This orbital structure corresponds to the intuitive notion of symmetry. Figure 44 shows the orbits of the standard representation with one orbit per line.

0000
0001 0010 0100 1000
0011 0101 1001 0110 1010 1100
0111 1101 1011 1110
1111

Figure 44. The orbits of the Standard Represntation.

The number and size of the orbits is preserved by conjugation, even though the vectors themselves may be different. Every conjugate of the standard representation will have five sets of orbits, two of size 1 (one of which will be 0000), two of size 4 and one of size 6.

We call the standard representation and its conjugates *Super-Class 0*. Sample orbits from the other eight super-classes of dimension 4 are given below. Two groups of matrices are conjugate only if their orbital structures are the same, but the converse is not necessarily true. The following is a list of the super classes and their orbital structures.

Super-Class 1:
0000
1110 1100 1011 1010 1001 1000 0111 0110 0101 0100 0011 0001
1101 0010
1111

Super-Class 2:
0000
1110 1101 1010 1001 0110 0101 0010 0001
1100 1011 1000 0111 0100 0011
1111

Super-Class 3:
0000
1111 1101 1011 1001 0111 0101 0011 0001
1110 1000 0100 0010
1100 1010 0110

Super-Class 4:
0000
1101 1100 1011 1010 0110 0001
1110 1001 0101 0100 0011 0010
0111
1000
1111

Super-Class 5:
0000
1110 1011 1010 0101 0100 0001
1101 0010
1100 1001 1000 0111 0110 0011
1111

Super-Class 6:
0000
1110 1101 1010 1001 0110 0101 0010 0001
1011 1000 0011
1100 0111 0100
1111

Super-Class 7:
0000
1100 1011 0110 0001
1000 0101 0010
1110 1001 0100 0011
1101 1010 0111
1111

Super-Class 8:
0000
1110 1101 1011 1010 1001 1000 0111 0110 0101 0100 0010 0001
1111 1100 0011

The characterization of strange symmetries for an arbitrary number of inputs is still an open problem.

7.2. Collapsed Symmetries

It is obvious that it is possible to represent S_n as a set of $n \times n$ matrices. It is also obvious that if $m > n$ it is possible to represent S_n as a set of $m \times m$ matrices. We start with the $n \times n$ representation and add $m - n$ rows and columns. We place 1s on the main diagonal of the new rows and columns, and zeros elsewhere. The resultant matrices are of the form shown in Figure 45, where M is the original matrix, I is the $(m-n) \times (m-n)$ identity matrix, and the zeros represent $n \times (m-n)$ and $(m-n) \times n$ matrices of all zeros. (The matrix Q is called the *direct product* of M and the identity matrix I.)

$$Q = \left(\begin{array}{c|c} M & 0 \\ \hline 0 & I \end{array} \right)$$

Figure 45. A Direct-Product Matrix.

For example, the group of Figure 46 is a 4×4 representation of S_3. The lines within the matrices are for clarification only.

$$\left(\begin{array}{ccc|c}1&0&0&0\\0&1&0&0\\0&0&1&0\\\hline0&0&0&1\end{array}\right)\left(\begin{array}{ccc|c}1&0&0&0\\0&0&1&0\\0&1&0&0\\\hline0&0&0&1\end{array}\right)\left(\begin{array}{ccc|c}0&1&0&0\\1&0&0&0\\0&0&1&0\\\hline0&0&0&1\end{array}\right)\left(\begin{array}{ccc|c}0&0&1&0\\0&1&0&0\\1&0&0&0\\\hline0&0&0&1\end{array}\right)\left(\begin{array}{ccc|c}0&1&0&0\\0&0&1&0\\1&0&0&0\\\hline0&0&0&1\end{array}\right)\left(\begin{array}{ccc|c}0&0&1&0\\1&0&0&0\\0&1&0&0\\\hline0&0&0&1\end{array}\right)$$

Figure 46. A Direct-Product Representation.

This matrix group is a subgroup of the standard representation of S_4 and represents the partial symmetry of the first three variables of a 4-input function.

What is not so obvious is that it is possible to create a representation of S_{n+1} using $n\times n$ matrices. This is done by extending the concept of the permutation matrix. Recall that a permutation matrix has exactly one 1 in each row and each column. Thus it is essentially a permutation of the rows of the identity matrix. If we start with the set of all $n\times n$ permutation matrices and add an additional row of all 1's we can create $(n+1)!$ matrices, and the result will be a closed matrix group representing S_{n+1}. This is fairly easy to see if consider the new matrices to be permutations of $n+1$ rows, the n that actually appear in the matrix, plus the missing row. Figure 47 shows a representation of S_3 in 2×2 matrices.

$$\begin{pmatrix}1&0\\0&1\end{pmatrix}\begin{pmatrix}0&1\\1&0\end{pmatrix}\begin{pmatrix}1&0\\1&1\end{pmatrix}\begin{pmatrix}0&1\\1&1\end{pmatrix}\begin{pmatrix}1&1\\1&0\end{pmatrix}\begin{pmatrix}1&1\\0&1\end{pmatrix}$$

Figure 47. An Extended Representation.

For S_4 there are two representations in 3×3 matrices, one with an extra row of all 1's and one with an extra column of all 1's. These two representations are *not* conjugate to one another.

We note in passing that we can create extended permutation matrices using real numbers if we replace the row or column of ones with a row or column of negative ones. Technically speaking, that is also what we have done here, since in GF(2) $1=-1$.

The pattern for S_4 can be repeated in the higher dimensions. Orbital analysis for these representations is straightforward. We use R_n to denote the representation of S_{n+1} in $n\times n$ matrices created by including a row of all ones, and C_n as the representation created by including a column of all ones. Let v_1 and v_2 be two n element vectors. Then v_1 and v_2 belong to the same orbit if and only if v_1 and v_2 have the same number of ones, or if v_1 has i ones and v_2 has j ones and $i+j=n+1$.

Thus the orbits of R_4 are as shown in Figure 48.

0000
0001 0010 0100 1000 1111
0011 0101 1001 0110 1010 1100 0111 1011 1101 1110

Figure 48. The Orbits of R_4

For C_n, first assume that the number of ones in v_1 is greater than or equal to the number of ones in v_2. Then, v_1 and v_2 are in the same orbit of C_n if and only if they have the same number of ones or if v_1 has $2i$ ones and v_2 has $2i-1$ ones for some $i < n$.

Thus the orbits of C_4 are as shown in Figure 49.

0000
0001 0010 0100 1000 0011 0101 1001 0110 1010 1100
0111 1011 1101 1110 1111

Figure 49. The Orbits of C_4

The groups R_n and C_n define *collapsed symmetry*. All known super-classes of symmetries can be constructed as the direct product of various collapsed symmetries. Because the orbital structure of R_4 is identical to that of C_4 one might suspect that they are conjugate to one another, and this is indeed the case. However, this is not true for all n.

The extended permutation matrices generate all collapsed symmetries for two and three input functions, but for four or more inputs there are additional collapsed symmetries. For four inputs, there is an additional 4×4 matrix group which is isomorphic to S_5, but not conjugate to R_4 or C_4. This group and its conjugates have two orbits, one containing 0000 and the other containing all non-zero vectors.

The number of collapsed symmetries for n inputs is probably related to the number of irreducible representations of S_{n+1} over GF(2), but the exact relationship is not known at this time.

8. Conclusion

We have shown how GF(2) matrices can be applied in a number of different contexts, but this work just scratches the surface. It is our belief that GF(2) matrices will eventually find applications in many diverse areas of EDA, and perhaps in other disciplines as well. The work in simulation is reasonably mature but much more work is needed in the area of logic synthesis. A drawback of using GF(2) matrices is the cost of computing them, but if they were applied earlier in the design cycle, it may be possible to get them for free. In other words, if a matrix is used to transform a set of inputs, it may be possible to encode data differently from the outset, thus avoiding the cost of the matrix entirely.

Areas that need more work are the use of singular matrices in function simplification, the exploration of exotic symmetries, and the application of matrices in high-level synthesis. There are undoubtedly many other application areas as well. We believe that the concept of single-bit matrices is crucial to the effective exploitation of GF(2) matrices in *any* context.

There are also some unsolved mathematical problems, like enumerating all matrix groups isomorphic to S_n for an arbitrary n.

Although the amount of work reported here is substantial, we believe that there is immeasurably more work to be done, much more than could be done by a single person or a single research team. We have a number of projects under way, and we hope that this paper will inspire much additional work in this area.

9. Appendix A, Proofs of Theorems.

9.1. Proof of Theorem 1.

Theorem 1. Any single-bit matrix is of order 2. That is, if A is a single bit matrix, then A^2 is the identity matrix.

Proof. Let $S_{i,j}$ be an $n \times n$ single-bit matrix. We will use the notation $a_{p,q}$ to denote the element in row p and column q of $S_{i,j}$ and $c_{p,q}$ to denote the corresponding element of $S_{i,j} \times S_{i,j}$. Now, consider the product $S_{i,j} \times S_{i,j}$. If $k \neq i$, then row k of $S_{i,j}$ has $a_{k,k} = 1$ and zeros elsewhere. Because row k of $S_{i,j} \times S_{i,j}$ is computed by multiplying row k of $S_{i,j}$ times the successive columns of $S_{i,j}$, row k of $S_{i,j} \times S_{i,j}$ is identical to row k of $S_{i,j}$. When computing row i of $S_{i,j} \times S_{i,j}$, the only non-zero elements of row i of $S_{i,j}$ are $a_{i,i}$ and $a_{i,j}$, so element $c_{i,k}$ will be computed using the formula $c_{i,k} = a_{i,i}a_{i,k} \times a_{i,j}a_{j,k}$. If $k \neq i$ and $k \neq j$ then $a_{i,k} = 0$ and $a_{j,k} = 0$ so $c_{i,k} = 0$. If $k = i$ then $c_{i,k} = c_{i,i} = a_{i,i}a_{i,i} \times a_{i,j}a_{j,i} = 1 \bullet 1 + 1 \bullet 0 = 1$. If $k = j$ then $c_{i,j} = a_{i,i}a_{i,j} \times a_{i,j}a_{j,j} = 1 \times 1 + 1 \times 1 = 0$. So the only non-zero element on row i of $S_{i,j} \times S_{i,j}$ is $c_{i,i}$, and $S_{i,j} \times S_{i,j}$ is the identity matrix.

9.2. Proof of Theorem 2.

Theorem 2. Suppose T is a non-singular $n \times n$ matrix with the first $i-1$ columns in upper triangular form, and that element i,i is zero. Then there exists a j and a single-bit matrix $S_{i,j}$ such that the first $i-1$ columns $S_{i,j} \times T$ are identical to those of T, and element i, j of $S_{i,j} \times T$ is equal to 1.

Proof. Since M is non singular, there must be a j such that element j,i of M is equal to 1. If this were not the case, rows i through n of M would be linearly dependent. We claim that $S_{i,j} \times M$ has the appropriate properties. We will designate the elements of $S_{i,j}$ as $a_{p,q}$, the elements of M as $b_{p,q}$ and the elements of $S_{i,j} \times M$ as $c_{p,q}$. All rows of $S_{i,j}$ except row i are identical to the corresponding rows of the identity matrix so, every row of $S_{i,j} \times M$ other than row i must be identical to the corresponding row of M. Row i of $S_{i,j}$ has two ones, one in position i and one in position j, so $c_{i,k} = a_{i,i}b_{i,k} + a_{i,j}b_{j,k}$. If $k < i$, then $c_{i,k} = a_{i,i}b_{i,k} + a_{i,j}b_{j,k} = 1 \times 0 + 1 \times 0 = 0$, so row i starts

with $i-1$ zeros. Thus the first $i-1$ columns of $S_{i,j} \times M$ are identical to the first $i-1$ columns of M. As for $c_{i,i}$, $c_{i,i} = a_{i,i}b_{i,i} + a_{i,j}b_{j,i} = 1\times 0 + 1\times 1 = 1$.

9.3. Proof of Theorem 3.

Theorem 3. Let M be a non-singular $n \times n$ matrix with the first $i-1$ columns in upper triangular form ($0 < i < n-1$), and with element i,i equal to 1. Let j be the smallest integer such that $j > i$ and element j,i is equal to 1. Then the first $i-1$ columns of $S_{j,i} \times M$ are identical to the first $i-1$ columns of M and column i of $S_{j,i} \times M$ is identical to column i of M except for element j,i, which is equal to 1 in M and equal to zero in $S_{j,i} \times M$.

Proof. Since all rows of $S_{j,i}$, except row j, are identical to the corresponding rows of the identity matrix, each row of $S_{j,i} \times M$ except row j is identical to the corresponding row of M. We need only concern ourselves with row j. Again, we designate the elements of $S_{j,i}$ as $a_{p,q}$, the elements of M as $b_{p,q}$, and the elements of $S_{j,i} \times M$ as $c_{p,q}$. Row j of $S_{j,i}$ has two 1's, one in position $a_{j,i}$ and one in position $a_{j.j}$. So $c_{j,k} = a_{j,i}b_{i,k} + a_{j,j}b_{j,k}$. If $k < i$ then $c_{j,k} = a_{j,i}b_{i,k} + a_{j,j}b_{j,k} = 1\bullet 0 + 1\bullet 0 = 0$. So the first $i-1$ columns of $S_{j,i} \times M$ are identical to the corresponding columns of M. For column i, $c_{j,i} = a_{j,i}b_{i,i} + a_{j,j}b_{j,i} = 1\bullet 1 + 1\bullet 1 = 0$.

9.4. Proof of Theorem 4.

Theorem 4. Let M be an upper triangular matrix. Let j be the largest integer such that column j of M has a 1 above the main diagonal, and let i be the largest integer such that $i < j$ and position i, j of M is equal to 1. Then the product matrix $S_{i,j} \times M$ is an upper triangular matrix that is identical to M except for position i, j which is equal to one in M and zero in $S_{i,j} \times M$.

Proof. Because every row of $S_{i,j}$, except row i, is identical to the corresponding row of the identity matrix, every row of $S_{i,j} \times M$, except row i, is identical to the corresponding row of M. We designate the elements of $S_{i,j}$ as $a_{p,q}$, the elements of M as $b_{p,q}$ and the elements of $S_{i,j} \times M$ as $c_{p,q}$. Row i of $S_{i,j} \times M$ is computed using the formula $c_{i,k} = a_{i,i}b_{i,k} + a_{i,j}b_{j,k}$. If $k < j$, then by the assumption of upper triangularity, $b_{j,k} = 0$, and $c_{i,k} = a_{i,i}b_{i,k} + a_{i,j}b_{j,k} = 1\bullet b_{i,k} + 1\bullet 0 = b_{i,k}$ If $k > j$, by the assumption that j is the largest integer such that column j of M has ones above the main diagonal, $b_{i,k} = b_{j,k} = 0$, and $c_{i,k} = b_{i,k} = 0$. If $k = j$ then $c_{i,j} = a_{i,i}b_{i,j} + a_{i,j}b_{j,j} = 1\bullet 1 + 1\bullet 1 = 0$.

9.5. Proof of Lemma 6.

Lemma 6. If $p \neq i$ then $c_{p,q} = b_{p,q}$.

Proof. Since the main diagonal of $S_{i,j}$ is all 1's, and since the only non-zero element of $S_{i,j}$ which is not on the main diagonal is $a_{i,j}$,

$$c_{p,q} =$$
$$a_{p,1}b_{1,q} + \cdots + a_{p,k}b_{k,q} \cdots + a_{p,p}b_{p,q} + \cdots + a_{p,r}b_{r,q} =$$
$$0 \bullet b_{1,q} + \cdots + 0 \bullet b_{k,q} \cdots + 1 \bullet b_{p,q} + \cdots + 0 \bullet b_{r,q} =$$
$$b_{p,q}$$

9.6. Proof of Lemma 7.

Lemma 7. If $p = i$ then $c_{p,q} = c_{i,q} = b_{i,q} + b_{j,q}$.

Proof: Since all elements of row i other than $a_{i,i}$ and $a_{i,j}$ are zero, and since $a_{i,i} = 1$ and $a_{i,j} = 1$

$$c_{p,q} = c_{i,q} =$$
$$a_{i,1}b_{1,q} + \cdots + a_{i,i}b_{i,q} \cdots + a_{i,j}b_{j,q} + \cdots + a_{i,r}b_{r,q} =$$
$$0 \bullet b_{1,q} + \cdots + 1 \bullet b_{i,q} \cdots + 1 \bullet b_{j,q} + \cdots + 0 \bullet b_{r,q} =$$
$$b_{i,q} + b_{j,q}$$

9.7. Proof of Theorem 8.

Theorem 8. In $S_{i,j} \times S_{m,n}$, $c_{i,j} = 1$ and $c_{m,n} = 1$.

Proof. If $i \neq m$, then $c_{m,n} = 1$ follows from lemma 6. Therefore, let us assume that $i = m$. Because $S_{i,j} \neq S_{m,n}$ it is necessary that $j \neq n$. In this case by Lemma 7, $c_{m,n} = c_{i,n} = b_{i,n} + b_{j,n} = b_{m,n} + b_{j,n} = 1 + b_{j,n}$. But since $i \neq j$ and $j \neq n$ $b_{j,n} = 0$, and $c_{m,n} = 1$. By Lemma 7, $c_{i,j} = b_{i,j} + b_{j,j} = b_{i,j} + 1$. However, since $i \neq j$ and $S_{i,j} \neq S_{m,n}$, $b_{i,j} = 0$ and $c_{i,j} = 1$.

9.8. Proof of Theorem 9.

Theorem 9. If $j \neq m$ then the main diagonal of $S_{i,j} \times S_{m,n}$ is all 1's, and the only non-zero elements off the main diagonal are the elements $c_{i,j}$ and $c_{m,n}$.

Proof. The only elements of $S_{i,j} \times S_{m,n}$ we need concern ourselves with are the elements of row i other than $c_{i,j}$ (and $c_{m,n}$ if $m = i$). First, $c_{i,i} = b_{i,i} + b_{j,i} = 1 + b_{j,i}$. Since $j \neq m$ and $j \neq i$, $b_{j,i} = 0$ and $c_{i,i} = 1$. Let $c_{i,q}$ be some element of row i other than $c_{i,i}$ $c_{i,j}$ or $c_{m,n}$. Then $c_{i,q} = b_{i,q} + b_{j,q}$ Since $q \neq i$ and since $c_{i,q}$ is not $c_{m,n}$, $b_{i,q} = 0$. By the same token, since $q \neq j$ and $j \neq m$, $b_{j,q} = 0$. So $c_{i,q} = b_{i,q} + b_{j,q} = 0 + 0 = 0$.

9.9. Proof of Theorem 10.

Theorem 10. If $j = m$ and $i = n$, then in $S_{i,j} \times S_{m,n}$, $c_{i,i} = 0$ and the only non-zero element of row i is $c_{i,j}$.

Proof. $c_{i,i} = b_{i,i} + b_{j,i} = b_{i,i} + b_{m,n} = 1+1 = 0$. Note that $m \neq i$, so $c_{m,n}$ is not in row i. Let $q \neq j$ and $q \neq i$. Then $c_{i,q} = b_{i,q} + b_{j,q}$. Since $i \neq j$, $b_{i,q} = 0$ and $b_{j,q} = 0$. Thus $c_{i,q} = 0$.

9.10. Proof of Theorem 11.

Theorem 11. If $j = m$ and $i \neq n$, then in $S_{i,j} \times S_{m,n}$, $c_{i,i} = 1$, $c_{i,j} = 1$ and $c_{i,n} = 1$. All other elements of row i are equal to zero.

Proof. Note that the conditions of the theorem imply that $c_{m,n}$ is not contained in row i, since this would mean that $m = i = j$, and it is necessary that $i \neq j$, so $m \neq i$. We already have $c_{i,j} = 1$ by Theorem 8. Now, $c_{i,i} = b_{i,i} + b_{j,i} = 1 + b_{j,i}$, but since $i \neq n$ and $i \neq j$, $b_{j,i} = 0$ and $c_{i,i} = 1$. Further, $c_{i,n} = b_{i,n} + b_{j,n} = b_{i,n} + b_{m,n} = b_{i,n} + 1$. But since $i \neq n$ and $m \neq i$, $b_{i,n} = 0$, and $c_{i,n} = 1$. If $q \neq i$ and $q \neq j$ and $q \neq n$, then $b_{i,q} = 0$ and $b_{j,q} = 0$. Therefore $c_{i,q} = b_{i,q} + b_{j,q} = 0 + 0 = 0$.

9.11. Proof of Theorem 12.

Theorem 12. The two single-bit matrices $S_{i,j}$ and $S_{m,n}$ commute with one another if and only if $j \neq m$ and $i \neq n$.

Proof. If $j \neq m$ and $i \neq n$ then by Theorem 1, $S_{i,j} \times S_{m,n}$ has ones on the main diagonal, ones in positions i, j and m, n, and zeros elsewhere. Also by Theorem 1, $S_{m,n} \times S_{i,j}$ has ones on the main diagonal, ones in positions m, n and i, j, and zeros elsewhere. Thus $S_{i,j} \times S_{m,n} = S_{m,n} \times S_{i,j}$. For the converse there are three cases. If $j = m$ and $i = n$, then by Lemma 1 $S_{i,j} \times S_{m,n}$ has a 1 in position j, j, and by Theorem 3 has a zero in position i, i. However $S_{m,n} \times S_{i,j}$ has a 1 in position $n, n = i, i$, hence $S_{i,j} \times S_{m,n} \neq S_{m,n} \times S_{i,j}$. If $j = m$ and $i \neq n$ then by Theorem 4, $S_{i,j} \times S_{m,n}$ has three ones off of the main diagonal, in positions i, j, m, n and i, n. However by Theorem 2, $S_{m,n} \times S_{i,j}$ has only two ones off of the main diagonal. Thus $S_{i,j} \times S_{m,n} \neq S_{m,n} \times S_{i,j}$. The case of $j \neq m$ and $i = n$ is similar.

9.12. Proof of Theorem 13.

Theorem 13. Let f be a function of n variables, and N be a singular $n \times n$ matrix such that any subset of size $n-1$ or smaller of the rows of N is linearly independent, but the sum of all n rows is the zero vector. Then N is of rank $n-1$ and f is compatible with N if and only if $f(v) = f(\overline{v})$ for all n element vectors v. (Note that $\overline{v}$ is the bit-wise complement of v.)

Proof. The rank of N follows immediately from the fact that any subset of $n-1$ rows of N is linearly independent. Now consider an n element vector v. Let $P = \{p_1, p_2, \ldots, p_k\}$ be the set of non-zero positions of v. $N(v)$ is obtained by computing the vector sum of the rows numbered $\{p_1, p_2, \ldots, p_k\}$ of N. Now consider the non-zero

positions of $\overline{v}$, $Q = \{q_1, q_2, \ldots, q_j\}$. By the definition of $\overline{v}$, $k + j = n$ and $Q = \{1, 2, \ldots, n\} - P$. Therefore $N(v) + N(\overline{v})$ is the sum of all rows of N and is equal to the zero vector. Now let $N(v) = (a_1, a_2, \ldots, a_n)$, and $N(\overline{v}) = (b_1, b_2, \ldots, b_n)$. Since $N(v) + N(\overline{v}) = 0$, it is necessary that $a_i = -b_i$ for all i. But in GF(2) $-x = x$ for all x and $a_i = b_i$ for all i, therefore $N(v) = N(\overline{v})$ for all v. Since the rank of N is $n-1$, if $N(v_1) = N(v_2) = N(v_3)$, then either $v_1 = v_2$, $v_2 = v_3$ or $v_1 = v_3$. Thus if $N(v_1) = N(v_2)$ then $v_2 = \overline{v}_1$. This implies that f is compatible with N if and only if $f(v) = f(\overline{v})$ for all v.

10. Appendix B. Symmetric Functions.

Spectrum	Minimal Sum-Of-Products Form
00000	0
00001	$abcd$
00010	$a'bcd + ab'cd + abc'd + abcd'$
00011	$abc + abd + acd + bcd$
00100	$a'b'cd + a'bc'd + a'bcd' + ab'c'd + ab'cd' + abc'd'$
00101	$a'b'cd + a'bc'd + a'bcd' + ab'c'd + ab'cd' + abc'd' + abcd$
00110	$abc' + a'bd + ab'd + acd' + bcd' + a'cd$
00111	$ac + ad + bc + bd + ab + cd$
01000	$a\,b'c'd' + a'b\,c'd' + a'b'c'd + a'b'cd'$
01001	$a\,b'c'd' + a'b\,c'd' + a'b'c'd + a'b'cd' + a\,b\,d$
01010	$a'bcd + ab'cd + abc'd + abcd' + a'b'c'd + a'b'cd' + a'bc'd' + ab'c'd'$
01011	$bcd + acd + abd + abc + a'b'c'd + a'b'cd' + a'bc'd' + ab'c'd'$
01100	$a'cd' + a'b'd + a'b\,c' + a\,c'd' + a\,b'c' + a\,b'd'$
01101	$a'cd' + a'b'd + a'bc' + ac'd' + ab'c' + ab'd' + abcd$
01110	$bc' + a'd + ab' + cd'$
01111	$a + b + c + d$
10000	$a'b'c'd'$
10001	$a'b'c'd' + abcd$
10010	$a'b'c'd' + a'bcd + ab'cd + abc'd + abcd'$
10011	$a'b'c'd' + bcd + acd + abd + abc$
10100	$a'b'c'd' + a\,b\,c'd' + a\,b'cd' + a\,b'c'd + a'b\,c'd' + a'b\,c\,d + a'b'cd$
10101	$a'b'c'd' + abc'd' + ab'cd' + ab'c'd + a'bcd' + a'bc'd + a'b'cd + abcd$
10110	$abc' + a'bd + ab'd + acd' + bcd' + a'cd + a'b'c'd'$
10111	$ac + ad + bc + bd + ab + cd + a'b'c'd'$
11000	$a'b'c' + a'c'd' + a'c'd' + b'c'd'$
11001	$a'b'c' + a'c'd' + a'c'd' + b'c'd' + a\,b\,d$
11010	$a'b'c' + a'c'd' + a'c'd' + b'c'd' + a'b\,c\,d + a\,b'cd + a\,b\,c'd + a\,b\,d'$
11011	$a'b'c' + a'c'd' + a'c'd' + b'c'd' + bcd + acd + abd + abc$
11100	$a'b' + c'd' + a'c' + a'd' + b'c' + b'd'$
11101	$a'b' + c'd' + a'c' + a'd' + b'c' + b'd' + a\,b\,d$
11110	$a' + b' + c' + d'$
11111	1

Table 1. The 4-Input Symmetric Functions.

Spectrum	Minimal Sum-Of-Products Form
00000	0
00001	$ab'cd'$
00010	$bc'd + ab'd$
00011	$bc'd + ab'd + ab'c$
00100	$bd' + ac'd' + a'cd'$
00101	$bd' + cd' + ad'$
00110	$bc' + ac' + bd' + ab'd + bcd'$
00111	$bc' + ab' + cd'$
01000	$a'b'd + b \text{◌̆}d$
01001	$a'b'd + b \text{◌̆}d + a \text{◌̆}cd'$
01010	d
01011	$d + ab'c$
01100	$a'c + bc + bd' + ac'd' + a'b'd$
01101	$a'c + bc + bd' + ad' + a'b'd$
01110	$b + d + a'c + ac'$
01111	$a + b + c + d$
10000	$a'b'c'd'$
10001	$a'b'c'd' + a \text{◌̆}cd'$
10010	$a \text{◌̆}d + b \text{◌́}d + a'b'c'd'$
10011	$a \text{◌̆}d + b \text{◌́}d + a \text{◌̆}c + a'b'c'd'$
10100	$a'b' + a'd' + b \text{◌̀}$
10101	d'
10110	$ac' + bc' + a'd' + bd' + ab'd$
10111	$d' + bc' + ab'$
11000	$a'b'c' + a'b'd + b \text{◌̆}d$
11001	$a'b'c' + a'b'd + b \text{◌̆}d + a \text{◌̆}cd'$
11010	$d + a'b'c'$
11011	$d + a'b'c' + a \text{◌̆}c$
11100	$a'b' + c'd' + bc$
11101	$d' + a'b' + bc$
11110	$a' + b + c' + d$
11111	1

Table 2. Conjugate Symmetric Functions

11. References

1. C.C. Tsai and M. Marek-Sadowska. “Detecting symmetric variables in boolean functions using generalized Reed-Muller forms.” *International Symposium on Circuits and Systems*, 1994.
2. C.C. Tsai and M. Marek-Sadowska, “Generalized Reed-Muller forms as a tool to detect symmetries,” *IEEE Trans. Comput.*, vol. 45, pp. 33-40, Jan. 1996.
3. R. Drechsler and B. Becker, “Sympathy: fast exact minimization of fixed polarity Reed-Muller expressions for symmetric functions,” *1995 European Design and Test Conference* p. 91
4. V. N. Kravets, K. A. Sakallah, “Generalized Symmetries in Boolean Functions,” *2000 International Conference on Computer-Aided Design (ICCAD '00)*, p. 526.
5. F. A. Aloul, Arathi Ramani, I. L. Markov, K. A. Sakallah “Solving difficult SAT instances in the presence of symmetry,” *Proceedings of the 39th conference on Design automation*, 2002, Pages: 731-736.
6. Maurer, P., “The Inversion Algorithm for Digital Simulation” IEEE Transactions on Computer Aided Design, Vol. 16, No. 7, July 1997, pp. 762-769.
7. Schuler, D., “Simulation of NAND Logic,” Proceedings of COMPCON 72, Sept 1972, pp. 243-5.
8. Breuer, M. A., A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, Woodland Hills, CA, 1976.
9. E. G. Ulrich, "Event Manipulation for Discrete Simulations Requiring Large Numbers of Events, " JACM, V.21, N.9, Sep. 1978, pp. 777-85.
10. Bryant, D. Beatty, K. Brace, K. Cho, T. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits," Proceedings of the 24th Design Automation Conference, 1987, pp. 9-16.
11. Appel, A. W., “Simulating Digital Circuits with One Bit Per Wire,” IEEE Transactions on Computer Aided Design, Vol. CAD-7, pp. 987-993, Sept., 1988.
12. Heydemann, M., D. Dure, “The Logic Automation Approach to Accurate and Efficient Gate and Functional Level Simulation,” *Proc. ICCAD-88*, 1988, pp. 250-253.
13. Lewis, D. M. “A Hierarchical Compiled Code Event-Driven Logic Simulator,” *IEEE Transactions on Computer Aided Design*, Vol 10, No. 6, pp.726-737, June 1991.
14. Olukotun, K., Heinrich, M., Ofelt, D., Digital system simulation: methodologies and examples, *Proceedings of the 35th conference on Design automation*, 1998, pp. 658-663.
15. Luo, Y., Wongsonegoro, T., Aziz, A., Hybrid techniques for fast functional simulation *Proceedings of the 35th conference on Design automation*, 1998, pp. 664–667.
16. Ganai, M., Aziz, A., Kuehlmann, A., Enhancing simulation with BDDs and ATPG, *Proceedings of the 36th conference on Design automation*, 1999, pp. 385-390.
17. Wilson, C., Dill, D., Reliable verification using symbolic simulation with scalar values, *Proceedings of the 37th conference on Design automation*, 2000, pp. 124-129.
18. Kölbl, A., Kukula, J., Damiano, R., Symbolic RTL simulation, *Proceedings of the 38th conference on Design automation*, 2001, pp. 47-52.
19. Cadambi, S., Mulpuri, C., Ashar, P., A fast, inexpensive and scalable hardware acceleration technique for functional simulation, *Proceedings of the 39th conference on Design automation*, 2002, pp. 570-575.

20. Schubert, K. Improvements in functional simulation addressing challenges in large, distributed industry projects, *Proceedings of the 40th conference on Design automation*, 2003, pp. 11-14.
21. Maurer, P., "Event Driven Simulation Without Loops or Conditionals," Proceedings of ICCAD-2000, 2000, pp. 23-26.
22. Maurer, P., "Logic simulation using networks of state machines," Proceedings DATE-2000, pp. 674-678, Mar. 2000.
23. Maurer, P., Efficient Event-Driven Simulation by Exploiting the Output Observability of Gate Clusters, IEEE Transactions on CAD, Vol. 22, No. 11, Nov., 2003, pp 1471-1486.
24. P. M. Maurer, "Using Conjugate Symmetry to Improve Simulation Performance," *Design Automation and Test in Europe Conference*, Mar 2006.
25. D. S. Robinson, *A Course in the Theory of Groups*, Springer, New York, 1995.
26. Burrow, Martin, *Representation theory of finite groups*, Academic Press, New York, 1965.
27. P. M. Maurer, "Metamorphic Programming: Unconventional High Performance," Computer, Vol. 37, No. 3, Mar, 2004, pp. 30-38.