Anti-Symmetry and Logic Simulation

Peter M. Maurer
Dept. of Computer Science
Baylor University
Waco, Texas 76798-7356

Abstract – Like ordinary symmetries, anti-symmetries are defined in terms of relations between function cofactors. For ordinary symmetries, two cofactors must be equal, for anti-symmetries two cofactors must be complements of another. It is known that ordinary symmetries can be used to improve simulation performance, but up until this point, it has not been known whether anti-symmetries could be used for this purpose. This paper shows that anti-symmetries can be as effective in boosting simulation performance as ordinary symmetries, sometimes even more so. Detailed detection and simulation algorithms are given along with a set of experimental results to show the effectiveness of the algorithms.

1 Introduction

Detecting symmetries in Boolean functions has proven to be a useful tool in many areas of electronic design automation. [1-7] There are many ways to define symmetric functions, but the fundamental idea behind all definitions is the concept of permutations. [8-10] In simplest terms, symmetric Boolean functions are those functions whose inputs can be rearranged in some fashion without changing the output value of the function, and permutations capture the idea of rearranging things.

The most important types of symmetry are *total symmetry* and *partial symmetry*. If a function is totally symmetric its inputs can be rearranged arbitrarily without changing the output of the function. If a function is partially symmetric, then one or more *subsets* of its inputs can be rearranged arbitrarily. Examples of totally symmetric and partially symmetric functions are $abcd$ and $abc+d$. Both partial and total symmetry can be defined in terms of permutations. There are many other types permutation-based symmetries (see [9, 11]) which are often lumped together and called *weak symmetries*. In this paper we will be concerned only with total and partial symmetry.

Total and partial symmetries are common in the circuits we encounter in practice. Both can be detected by examining pairs of variables. Two variables constitute a *symmetric variable pair* if they can be exchanged without altering the output of the function. Symmetric variable pairs are transitive. If a can be exchanged with b and b can be exchanged with c, then a can be exchanged with c. A function is totally symmetric if and only if every pair of input variables is symmetric. A function is partially symmetric in the variables $x_1,...,x_k$ if x_i and x_j is a symmetric variable pair for all i and j with $1 \le i < j \le k$.

Suppose f is an *n*-input Boolean function with input variables $x_1,...,x_n$. The cofactors of f with respect to x_1 are the functions $f_{0x...x}$ and $f_{1x...x}$, where $x...x$ represents a string of $n-1$ x's. The functions $f_{0x...x}$ and $f_{1x...x}$ are computed by setting the variable x_1 to 0 and 1 in f. The exact procedure for computing a cofactor depends on the representation of the function. Cofactors can be computed with respect to any variable, or with respect to a set of variables. When there is no opportunity for confusion, we omit the x's and simply place the 1's and 0's in the subscript. In symmetry detection, it is common to compute cofactors with respect to pairs of variables. There are four such cofactors f_{00}, f_{01}, f_{10}, and f_{11}. These cofactors are tested for equality to detect symmetric variable pairs. Other relations between these cofactors can be used to define new types of symmetry. [12, 13]

In addition to permutations, there are other types of transformations that can be used to define symmetries. The most common of these is the transformation $N_{a,...,k}$ where the subscripts a through k are unique numbers in the range 1 through n. The transformation $N_{a,...,k}$ indicates the inversion of those variables in the range x_1 through x_n whose subscripts appear in the list $a,...,k$. These transformations can be used to define the multi-phase symmetries, but these symmetries can also be defined through cofactor relations.

One of the latest developments in symmetry classification is matrix-based symmetry. [7] All permutations can be specified as non-singular matrices, but not all non-singular matrices can be specified as permutations. Thus matrix-based symmetry is an extension of permutation-based symmetry. Conjugate symmetry is one form of matrix-based symmetry, some forms of which can be detected using cofactor-relations.

The algorithms discussed in this paper are based on cofactor relations. The two most important types of relations are the classical (or ordinary) relations and the anti-relations, [14] which we will discuss in the next two sections.

2 Classical Symmetry

The classical relations are defined in terms of the two-variable cofactors of a function, which we designate: f_{00}, f_{01}, f_{10}, and f_{11}. The cofactors of a function are obtained by setting one or more variables of the function to constant values. When it is necessary to distinguish the precise variable in question, a subscript specifying the positions of the variables is used. An example of such a specification is $f_{xx1xx0xx}$, which denotes an 8-input function with the third and sixth variables set to 1 and 0 respectively. When speaking without reference to specific variables, we will omit the x's, and use the abbreviated form f_{10}.

The classical relations are equality relations between the four cofactors f_{00}, f_{01}, f_{10}, and f_{11}. There are six possible relations, each of which represents a certain type of symmetry between a pair of variables. These relations are given in Figure 1, along with their respective symmetry types. The last four relations in Figure 1 are often lumped together, but in practice they are distinct in the sense that different comparisons must be performed to detect them and different procedures must be used to handle them.

In a sense, the only relation that truly represents symmetry is $f_{01} = f_{10}$, ordinary symmetry. The other relations represent variable pairs that are *not* symmetric, but can be "corrected" to become symmetric. When an ordinary symmetry is detected (before or after "correction"), the symmetric variables are combined into a single "clustered" variable that can subsequently be treated as an ordinary variable. A simulator that makes use of all six classical relations is described in [7]. For completeness, we describe the handling of these relations here. Our simulator is based on that described in [7], but is able to handle more types of relations.

Relation	Symmetry Type
$f_{01} = f_{10}$	Ordinary
$f_{00} = f_{11}$	Multi-Phase
$f_{01} = f_{11}$	Single-Variable A
$f_{10} = f_{11}$	Single-Variable B
$f_{10} = f_{00}$	Multi-Phase Single-Variable A
$f_{01} = f_{00}$	Multi-Phase Single-Variable B

Figure 1. Cofactor Relations.

The multi-phase relation, $f_{00} = f_{11}$, indicates that the function is symmetric as long as one variable is inverted with respect to the other. It is possible to treat the multi-phase relation as ordinary symmetry after adding a NOT gate to one of the inputs. As pointed out in [15] it is necessary to transform the *state-space* (in effect, the truth table) of the function when treating multi-phase symmetry as ordinary symmetry. The state-space transformation modifies the truth-table of the function to make it symmetric, while inverting the input variable compensates for the state-space transformation.

The single-variable relations represent two types of conjugate symmetry. Not all conjugate symmetries manifest themselves as single-variable symmetries, but the mechanisms used to detect single-variable symmetries can be extended to detect most, if not all conjugate symmetries. As with multi-phase symmetries, it is possible to treat conjugate symmetries as if they were ordinary symmetries using a state-space transformation along with a corrective input transformation. In this case the corrective input transformation consists of a collection of XOR gates on the inputs of a function. These XOR gates compute a matrix transformation of the inputs prior to passing the inputs into the function. The input transformation is similar to the first layer of logic in some forms of three-level minimization. [16, 17]

The multi-phase single-variable relations represent the combination of conjugate symmetry and multi-phase symmetry. The state-space transformations and the corrective transformations can be combined without interference so these types of symmetry can be handled by combining the techniques for multi-phase and conjugate symmetry.

The result of symmetry detection is a multi-dimensional state machine which represents the state of a Boolean function. (In a sense, the state of a Boolean function is the input for which it was last evaluated, as long as we treat clustered variables properly.) Each dimension of the state machine represents a cluster of symmetric variables. It is convenient to think of the multi-dimensional machine as an extended type of hypercube with several states along each dimension. For a simple, non-clustered input variable, the dimension will have two states representing input values of zero and one. For a cluster of n variables, the dimension will have $n+1$ states with the state representing the number of ones in the n inputs. See Figure 2 for an example. Figure 2 illustrates a function with a simple variable A, and a clustered variable containing three simple variables, B, C, and D.

Another way to view the *n*-dimensions of a gate state-machine is as a collection of *n* input-ports. For ordinary and multi-phase symmetries, there is a one-to one mapping between input ports and clusters of function inputs. For conjugate symmetry, an event on an input can generate events on many different ports. This technique is used to compute the XOR functions on the inputs. We will explain this further in Section 4.

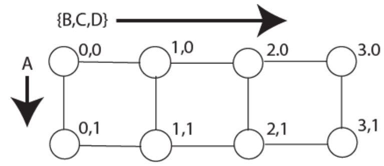

Figure 2. A Gate State-Machine.

3 Anti-Symmetry

Anti-symmetry is known by several other names, including *skew symmetry* and *negative symmetry*. Anti-symmetry is based on the observation that relations of the form $f_{01} = f_{10}$ can be written $f_{01} \oplus f_{10} = \overline{0}$, where $\oplus$ represents the XOR operation and $\overline{0}$ represents the constant zero function. If we reformulate our six relations and replace the constant zero with the constant one function, $\overline{1}$, we obtain the six anti-symmetry relations given in Figure 3.

Relation	Anti-Symmetry Type
$f_{01} \oplus f_{10} = \overline{1}$	Ordinary
$f_{00} \oplus f_{11} = \overline{1}$	Multi-Phase
$f_{01} \oplus f_{11} = \overline{1}$	Single-Variable A
$f_{10} \oplus f_{11} = \overline{1}$	Single-Variable B
$f_{10} \oplus f_{00} = \overline{1}$	Multi-Phase Single-Variable A
$f_{01} \oplus f_{00} = \overline{1}$	Multi-Phase Single-Variable B

Figure 3. The Anti-Symmetry Relations.

As with multi-phase and conjugate symmetries, anti-symmetries can be transformed into ordinary symmetries using state-space transformations. These transformations are easier to visualize if we place the four cofactors f_{00}, f_{01}, f_{10} and f_{11} into a hyper-linear structure as shown in Figure 4. The grayed functions are inverted.

There are several different state-space transformations that will transform an anti-symmetry into an ordinary symmetry. The simplest is to complement one of the cofactors f_{01}, or f_{10}. If $f_{01} \otimes f_{10} = \overline{1}$ is true, then $f'_{01} \otimes f_{10} = \overline{0}$ and $f_{01} \otimes f'_{10} = \overline{0}$. These corrective actions are shown in Figure 4. However, this naïve transformation results in a complex corrective action to undo the transformation. Let us suppose that an ordinary anti-symmetry is found between the variables are a and b and that f_{01} has been complemented to transform the symmetry into a classical symmetry. This means that when the transformed function is evaluated, the output will be inverted whenever $a = 0$ and $b = 1$. The corrective function shown in Figure 5 is applied during the simulation process to produce the correct function output.

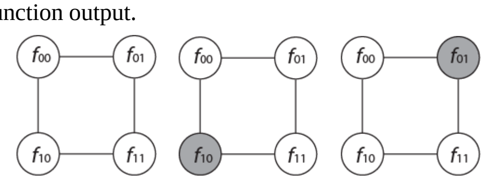

Figure 4. Naïve Corrective Actions.

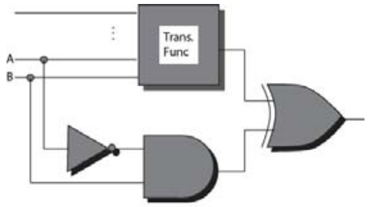

Figure 5. A Naïve Corrective Function.

If a function has several anti-symmetric variable pairs, the corrective function can become complicated with a NOT and an AND gate required for every pair. (Only a single XOR gate is required regardless of how many anti-symmetries are found.) Even though our simulator is able to eliminate the NOT gates, the proliferation of AND gates may negate the benefit of detecting the anti-symmetry.

To simplify the corrective procedure, we use an over-kill method when transforming the function. Instead of just complementing f_{01} (or f_{10}) we also complement f_{11}, as shown in Figure 6. This means that the output of the transformed function is inverted whenever $b=1$ (or $a=1$). This eliminates the AND and NOT gates, as shown in Figure 7. Regardless of how many anti-symmetric variable pairs are detected for a function, only a single XOR gate is required on the output. This XOR gate must have one input for each detected anti-symmetric pair. In Section 5 we will show how we get the XOR gate for free.

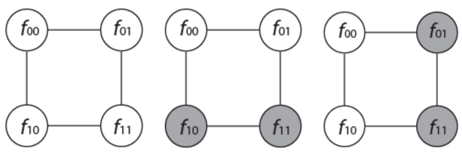

Figure 6. Sophisticated Corrective Actions.

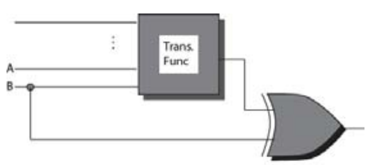

Figure 7. A Simple Corrective Function.

4 Symmetry Detection

The corrective function for anti-symmetries, regardless of how many are found for a particular function, is identical to that of Figure 7. There is a single XOR gate which has the transformed function as an input. In addition, several of the input variables of the function are fed directly into the XOR gate. In determining the additional inputs of the XOR gate there are several problems that must be addressed. When combining anti-symmetry with conjugate symmetry, adding inputs to the correcting XOR becomes more complicated. We need to determine how to proceed in these circumstances. We also need to determine what should happen when a variable is added twice to the correcting XOR. And finally we need to determine how to detect anti-symmetry with respect to clustered variables.

Clustered variables have been discussed thoroughly in [7, 15, 18], but only with respect to classical symmetries. Ordinary symmetries can be detected using structures such as that pictured in Figure 2, by examining cofactors along the anti-diagonals In this case node (1,0) must equal (0,1), (2,0) must equal (1,1) and (3,0) must equal (2,1). If there is more than one clustered variable, there may be several vertices along each diagonal, all if which must be equal. If there are more than two dimensions, the diagonal tests must be repeated for each of the 2^{n-2} planes containing the two variables. There is no required relationship between separate diagonals or separate planes.

Multi-phase symmetry can be detected by reversing the structure along one dimension and then testing for ordinary symmetry. Conjugate symmetry can be detected by reversing the odd numbered rows (starting with 0 at the top) or the odd numbered columns (starting with 0 at the left), and then testing for ordinary symmetry. Combined multi-phase and conjugate symmetry is detected by reversing the even numbered rows and columns. (In practice this is accomplished by reversing the entire structure and then reversing the odd rows or columns.) The reversals can be done without altering the structure by indexing rows, columns, or the entire dimension in reverse order.

The symmetry detection algorithm is based on that described in reference [7]. Consider the left-most state machine of Figure 8. This state machine represents a 4-input function with two clustered variables. Assume that the inputs to the function are a, b, c, and d. and have been clustered into two pairs (a,b) and (c,d). The horizontal dimension of the state machine represents the state of the pair (a,b), while the vertical dimension represents the state of the pair (c,d). The state value is either 0, 1, or 2, depending on how many variables of the pair are set to 1.

To detect symmetries between the clustered variables (a,b) and (c,d) it is necessary to examine the reverse diagonals. If the states with the same letter (L, F, and G) contain the same function, then the two clustered variables (a,b) and (c,d) are symmetric with one another.

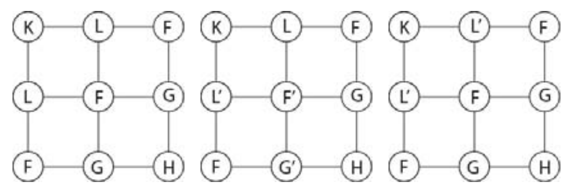

Figure 8. Clustered-Variable State Machine.

When an anti-symmetry exists between any two variables in two different clustered pairs, then an anti-symmetry exists between every pair of variables in the two of clustered variables. This implies that a function must alternate with its complement along each back-diagonal. This condition is shown in the middle state-machine of Figure 8. (See [15] for more detail.)

To convert the anti-symmetry into an ordinary symmetry, we invert the functions in the odd-numbered rows or the odd-numbered columns, as shown in the third state machine of Figure 8. This is where the XOR of the individual variables in the clustered variable takes the value 1. Thus to correct for the inverted column of Figure 8, we must add the variables *a* and *b* to the corrective XOR gate as shown in Figure 9.

Regardless of how many anti-symmetries are found, only a single XOR gate is required for the corrective function. When the first anti-symmetry is detected, we add the XOR gate to the output of the function. We also create a list of the input variables that must be added to the inputs of the XOR gate. When new anti-symmetries are detected, new input variables are added to the list.

It is sometimes necessary to add the same input to the list twice. Since the two inputs are identical, the pair will have either the value (0,0) or the value (1,1). In both cases, the XOR of the two values is 0, which will not change the value computed from the other inputs and both duplicated inputs can be removed. Thus, when we add an input to the list, we check to see whether it is currently on the list. If so, then we remove it instead of adding it.

When anti-symmetry is combined with conjugate symmetry we have an additional problem. The function has n inputs and the state-machine has n input ports, but with conjugate symmetry, the mapping between inputs and ports is not one-to-one. Several function inputs can be directed into a single port, and a single function input can be directed into several ports. Since symmetry detection is done with respect to ports, and correction is done with respect to function inputs, it is necessary to maintain a mapping between the two. The symmetry detection algorithm maintains a mapping of function-input to port mappings. When correction must be done with respect to a port, every function input directed into that port is added to (or removed from) the list of corrected function inputs. (The matrix is also used to create the corrective XOR functions at the state-machine inputs.)

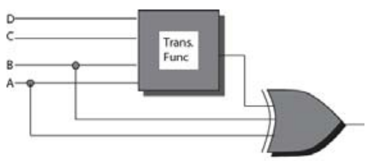

Figure 11. A Clustered Corrective Function.

To avoid conflicts with multi-phase and conjugate symmetry we take note of the reversing operations used during the detection process. To detect multi-phase symmetry it is necessary to reverse either all rows or all columns, but not both. If the rows have been reversed, then we transform the function by inverting the odd rows. If the columns have been reversed then we invert the odd columns. By placing the reversal and the inversion along the same dimension, we invert the same vertices that would have been inverted for an ordinary symmetry.

Conjugate symmetry is handled the same way. If the odd rows are reversed, we invert the odd rows. If the odd columns are reversed, we invert the odd columns. The algorithm for symmetry detection is given in section S1, Figure S1.

The algorithm for detecting symmetric variable pairs is virtually identical for the two types of symmetry, ordinary symmetry and anti-symmetry. State-space transformations are used to combine ordinary and anti-symmetry with conjugate and multiphase symmetry. The basic algorithm is given in Section S1, Figure S2. This algorithm is executed twice, once for anti-symmetry and once for ordinary symmetry.

Checking the back diagonals is done by selecting each diagonal, and obtaining a comparator function from the first element of the diagonal. The comparator function is compared to the function contained in the other vertices along the diagonal. For anti-symmetry, two comparator functions are obtained. The first is taken from the first vertex of the diagonal and the second is obtained by inverting the first function. These functions are compared in alternating fashion along the diagonal. The two algorithms are given in Section S1, Figures S3 and S4.

5 Simulation Code

Detecting the various types of symmetry is beneficial to our simulator is because it can operate faster with symmetric functions. It is possible for the corrective functions to negate the benefit of detecting the certain types of symmetries, but as we will show in this section, we can either eliminate, or greatly reduce the cost of these functions.

Our simulator is a compiled simulator. Simulation detection is done during the code generation phase so that the cost of symmetry detection can be amortized over many simulations. The generated code is compiled to create a simulator for the circuit. Thus, when we report simulation times, these times do not include the cost of symmetry detection. Symmetry detection is extremely fast and takes less than a second for all but one circuit.

At run time, each function is implemented of an *n*-dimensional state machine, with the current state represented as an *n*-element vector giving the indices of the current state. A simple input can either increment or decrement a single index by 1. If $A = \{a_1, ..., a_k\}$ is a clustered variable, then each of the simple inputs, $a_1, ..., a_k$, increments and decrements the same index. A separate index is used for each dimension of the state machine, regardless of whether the dimension represents a clustered variable or simple variable.

The operations performed with respect to simple variables alternate with one another. If the current action is to increment the index, then the next action will be to decrement it, and vice versa. It is not necessary to maintain the value of the input as long as we know which operation to perform next. The NOT gates required by multi-phase symmetry can be eliminated as long as the increment/decrement state of the input is initialized properly. Thus we get multi-phase symmetry for free.

Suppose we have two successive events for the same input. The first event will increment (or decrement) an index, and the second event will decrement (or increment) the same index, leaving the state unchanged. In effect we have computed the “exclusive or” of the two events. Since this exclusive-or is inherent in the state machine, we can get the corrective layer of XOR gates required by conjugate symmetry almost for free. Without conjugate symmetry, there is a one-to-one correspondence between the input-ports of a state machine and clusters of function inputs. When implementing functions with conjugate symmetry, we break this one-to-one correspondence, and permit input variables to activate events on more than one input port. The successive events are XORed with one another computing the required XOR function at the cost of queuing additional events. (To save time, we actually queue a single multi-action event rather than multiple events.)

The state of each input port is recorded as 1 or -1, depending on whether the next operation is an increment or decrement. The data structure representing the event has an array of pointers to port states and port indices. When an event is executed, the port state is added to the port index and is then negated. The indices are used to determine whether the output of the function has changed. If the output changes then an event is queued. If an event is already queued for the output, the queued event is cancelled.

Surprisingly enough, we can get the anti-symmetry correction essentially for free. Since events are processed one at a time, we need only concentrate on the effect of one event. Referring back to Figure 7, suppose an event on input *B* causes the output of the transformed function to change. Any event on a single input of an XOR gate causes the output of the gate to change, so this event will be propagated to the output of the XOR. This will queue a series of events for the original function output. However, this same event will propagate directly to the input of the XOR gate, causing the previously queued events to be cancelled. Thus, if the output of the transformed function changes, no output events will be scheduled.

Suppose now that an event occurs on input B, but the output of the transformed function *does not change*. Because the output does not change, no event propagates into the XOR gate from the function output. However, the event on B still propagates directly into the XOR gate, causing events to be scheduled for the output of the XOR, which represents the output of the original function.

The correcting XOR gate causes the usual effect of an event to be reversed. Events propagate when the output of a function does not change, and no event propagates when the output changes. We generate two sets of run-time routines. The first set contains a comparison of the form “if Old = New then propagate” while the second contains a comparison of the form “if Old $\neq$ New then propagate”. The first set is used for those inputs directed into the corrective XOR, the second is used for other inputs. These two routines are virtual functions that are called through function pointers determined at compile time. No run-time test is required to distinguish the two types of inputs. Effectively, the corrective XOR is obtained for free.

Figures S5 and S6 give the run-time code for processing an event. Our simulator does not require gate simulation code, so the algorithms of Figures S5 and S6 represent virtually the entire run-time code of our simulator.

When examining these algorithms, it is important to remember that as much decision-making as possible is done at code generation time. The choice between the different event-handlers is done at compile time, as is the index resolution for GateState[i]. The Value[GateState] index must be computed at run time, along with the test between new and old gate values. Input port mapping and initial states are computed at code generation time.

6 Experimental Results

For our experimental results, we used the ISCAS 85 benchmarks. [19] Although these benchmarks are the *de facto* standard for determining simulation performance, they are specified at the gate level rather than at the Boolean function level. In this respect, they are not the most ideal vehicle for determining the effectiveness of our simulator, however they exhibit a wide variety of symmetries of all types.

The main problem with gate-level circuits is that one must attempt to reconstruct the Boolean functions from which the circuit was created. This is a decidedly non-trivial task. As an approximation to this we first identify the fanout-free networks in the circuit. These networks represent single-output functions, but are only an approximation of the original Boolean functions. About 50% of the gates in each circuit end up in single-gate fanout-free networks. We do not apply our algorithm to single gates, because their symmetries are already well-known.

A few circuits have very large fanout-free networks which represent several Boolean functions combined into a single network. To break these giant networks down into something resembling real Boolean functions we limit the number of inputs of a single network. We have experimented with different limits and have found that a limit of eight tends to expose the most symmetries. The limit is only approximate. It is possible for an individual partition to have more than eight inputs.

Our first experiment was to determine the number of anti-symmetries that appear in these circuits. Figure 12 gives the number of anti-symmetries found in each circuit when no other types of symmetries are detected. The numbers are counts of anti-symmetric variable pairs. These are further broken down into ordinary anti-symmetries (Ord.), multi-phase anti-symmetries (M.P.), conjugate anti-symmetries (Conj.), and combined multi-phase/conjugate symmetries (C.M.P). Every one of the ten benchmarks contains some anti-symmetries. The total ranges from a low of 10 for c432, to a high of 944 for c6288. This experiment verifies that anti-symmetries are indeed prevalent enough to be worth pursuing. If we examine the breakdown of sub-types, it is clear that it is necessary to combine anti-symmetry with multi-phase and conjugate symmetry. Note, for example, benchmark c499, which has no ordinary anti-symmetries, but has 104 symmetric pairs of other types.

To compare the prevalence of anti-symmetries with that of classical symmetries we determined the number of classical symmetries in each of the four categories. The results of this experiment are given in Figure 13. For most of the circuits there are significantly more classical symmetries than anti-symmetries, but there are two notable exceptions. Both c1355 and c6288, have more anti-symmetries than classical symmetries. Clearly we should detect anti-symmetries to handle these circuits effectively.

When combining the detection of anti-symmetries with that of classical symmetries, we encounter the phenomenon known as “symmetry masking.” This occurs when the detection of one type of symmetric pair prevents the detection of a different type. This is not necessarily a problem, since the two symmetries might be with respect to the same pair of variables. Nevertheless, it is not possible to add the results of Figure 12 to those of Figure 13 to determine the total number of symmetric pairs. When symmetry masking occurs, it is important to detect the “simplest” types of symmetries first. (The “simplest” are those with the most efficient corrective operations.)

Ckt	Ord.	M.P.	Conj.	C.M.P.	Total
c432	10	0	0	0	10
c499	0	40	64	0	104
c880	5	12	18	2	37
c1355	104	104	0	0	208
c1908	14	2	99	0	115
c2670	14	11	72	2	99
c3540	173	11	52	9	245
c5315	29	46	139	22	236
c6288	480	464	0	0	944
c7552	53	126	310	44	533

Figure 12. Anti-Symmetries

Ckt	Ord.	M.P.	Conj.	C.M.P.	Total
c432	47	45	9	2	103
c499	110	24	40	0	174
c880	133	4	13	2	152
c1355	14	24	0	104	142
c1908	138	4	4	0	146
c2670	279	10	11	58	358
c3540	198	169	74	23	464
c5315	298	12	120	239	669
c6288	0	0	480	0	480
c7552	710	16	108	119	953

Figure 13. Classical Symmetries.

A complete study of detection order is beyond the scope of this paper. For this paper we first search for classical symmetries in the order: Ordinary, Multi-Phase, Conjugate, Conjugate/Multi-Phase. Then we search for anti-symmetries in the same order. The results of detecting both classical and anti-symmetries are given in Figure 14. In some cases, the detection of anti-symmetries suppresses the detection of a few classical symmetries, but the detection of additional symmetries compensates for this. In some cases, many of the anti-symmetries are masked by the classical symmetries. In other cases the number of detected anti-symmetries significantly adds to the total number of symmetries.

To determine the effect of symmetry detection on simulation performance, we compared four different simulations for each benchmark circuit. The four simulations are with no symmetry detection, with anti-symmetry alone, with classical symmetry alone, and with combined classical and anti-symmetry. Each simulation was performed on a dedicated 3.06 Ghz Xeon processor with 2GB of 233 Mhz memory. The results, which are shown in Figure 15, are in seconds of execution time for 500,000 input vectors.

All of our simulations are compiled simulations, meaning that the simulator generates code which is then compiled and executed in stand-alone fashion. The compile step, which includes all symmetry detection, took less than a second for each circuit, regardless of the types of symmetries detected. The run times given in Figure 15 do not include the time required to detect symmetry.

Ckt	Classical	Anti	Total
c432	103	0	103
c499	174	0	174
c880	152	19	171
c1355	142	104	246
c1908	146	0	146
c2670	357	34	357
c3540	463	5	468
c5315	664	32	696
c6288	480	464	944
c7552	951	99	1050

Figure 14. Classical and Anti-Symmetries

Ckt	None	Anti	Classical	Combined
c432	2.54	2.52	2.35	2.28
c499	3.75	3.43	3.20	3.20
c880	5.54	5.49	4.85	4.78
c1355	5.30	4.91	5.17	5.08
c1908	5.12	5.01	5.04	4.95
c2670	17.77	17.57	17.06	16.97
c3540	12.37	12.20	11.18	11.20
c5315	26.71	26.23	24.11	24.12
c6288	18.81	17.29	19.30	18.47
c7552	31.38	30.73	29.09	29.12

Figure 15. Running Times.

In most cases, the combined anti and classical symmetries gave the best performances, but there are some exceptions, most notably c6288. This circuit has 944 anti-symmetries of the ordinary and multi-phase type. It also has 480 classical symmetries of the conjugate type. The conjugate symmetries are slightly more costly than the other types, and in c6288 they are distributed through 480 subcircuits, one per subcircuit. With this sparsity, the additional cost of the conjugate symmetry outweighs the concomitant simplification of the state machine. On the other hand, the anti-symmetries are all of the ordinary and multi-phase type, both of which are essentially free. Thus for c6288, the best running time is observed for anti-symmetries alone. The circuit c1355 also showed the best performance for anti-symmetries alone.

7 Conclusion

Even though detecting anti-symmetries is somewhat more difficult than detecting classical symmetries, the detection can be done quickly, virtually always with beneficial results. There are some problems yet to be resolved, most notably determining the most effective ordering of symmetry types.

There are other types of relations that may permit detection of even more symmetries, with a concomitant simplification of gate state machines. These are the Kronecker relations [13] which involve three or more cofactors, and the general relations of the form $f_{01} \oplus f_{10} = g$, where g is some function other than constant one or constant zero. Regardless of whether these other relations prove to be beneficial, detecting and handling anti-symmetry is clearly a useful tool for improving simulation performance.

8 References

1. C. E. Shannon, "The synthesis of two-terminal switching circuits," *Bell System Technical Journal*, Vol.28, *No.1*, pp. 59-98, 1949.
2. C. R. Edwards and S. L. Hurst, "A digital synthesis procedure under function symmetries and mapping methods," *IEEE Transactions on Computers*, Vol.27, *No.11*, pp. 985-997, 1978.
3. D. Moller, P. Molitor, R. Drechsler and J. W. G. U. Frankfurt, "Symmetry based variable ordering for ROBDDs," *IFIP Workshop on Logic and Architecture Synthesis*, pp. 47-53, 1994.
4. C. Scholl, D. Moller, P. Molitor and R. Drechsler, "BDD minimization using symmetries," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol.18, *No.2*, pp. 81-100, 1999.
5. T. Sasao, "A new expansion of symmetric functions and their application to non-disjoint functional decompositions for LUT type FPGAs," *IEEE International Workshop on Logic Synthesis*, pp. 105-110, 2000.
6. V. N. Kravets and K. A. Sakallah, "Constructive library-aware synthesis using symmetries," *Design Automation and Test in Europe*, pp. 208-213, 2000.
7. P. M. Maurer, "Conjugate Symmetry," *Formal Methods Syst. Des.*, Vol.38, *No.3*, pp. 263-288, 2011.
8. V. N. Kravets and K. A. Sakallah, "Generalized symmetries in boolean functions," *IEEE International Conference on Computer Aided Design*, pp. 526-532, 2000.
9. J. Mohnke, P. Molitor and S. Malik, "Limits of using signatures for permutation independent Boolean comparison," *Formal Methods Syst. Des.*, Vol.21, *No.2*, pp. 167-191, 2002.
10. P. M. Maurer, "An application of group theory to the analysis of symmetric gates," Department of Computer Science, Baylor University, Waco, TX 76798, http://hdl.handle.net/2104/5438, 2009.
11. V. N. Kravets and K. A. Sakallah, "Generalized symmetries in boolean functions," Advanced Computer Architecture Laboratory Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109, http://web.eecs.umich.edu/techreports/cse/00/CSE-TR-420-00.pdf, 2002.
12. C. C. Tsai and M. Marek-Sadowska, "Generalized Reed-Muller forms as a tool to detect symmetries," *IEEE Transactions on Computers*, Vol.45, *No.1*, pp. 33-40, 1996.
13. M. Chrzanowska-Jeske, A. Mishchenko and J. R. Burch, "Linear Cofactor Relationships in Boolean Functions," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol.25, *No.6*, pp. 1011-1023, 2006.
14. C. C. Tsai and M. Marek-Sadowska, "Boolean functions classification via fixed polarity Reed-Muller forms," *IEEE Transactions on Computers*, Vol.46, *No.2*, pp. 173-186, 1997.
15. P. M. Maurer, "Extending symmetric variable-pair transitivities using state-space transformations," Department of Computer Science, Baylor University, Waco, Texas 76798, http://hdl.handle.net/2104/8185, 2011.
16. F. Luccio and L. Pagli, "On a new Boolean function with applications," *IEEE Transactions on Computers*, Vol.48, *No.3*, pp. 296-310, 1999.
17. A. Bernasconi, V. Ciriani, F. Luccio and L. Pagli, "Synthesis of Autosymmetric Functions in a New Three-Level Form," *Theory of Computing Systems*, Vol.42, *No.4*, pp. 450-464, 2008.
18. P. M. Maurer. Efficient event-driven simulation by exploiting the output observability of gate clusters. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* Vol.22, *No.11*, pp. 1471-1486.
19. F. Brglez, P. Pownall and R. Hum. Accelerated ATPG and fault grading via testability analysis. Presented at Proceedings of IEEE Int. Symposium on Circuits and Systems.

9 S1. Pseudo-Code

In this section we present the pseudo-code for all algorithms. Some of these algorithms have been described in [7], but for completeness, are included here in somewhat different form.

```
Compute f00, f01, f10 and f11 and place them in a 2-dimensional
    hypercube structure.
Repeat
  Test the back diagonals for ordinary symmetry
  Test the back diagonals for anti-symmetry
  If a symmetric variable pair is detected
    Collapse the structure to one less dimension by combining
      vertices along the back diagonals.
    Add any required corrective functions
  Endif
  If uneliminated variables remain
    Compute two new cofactors from all existing cofactors
      Double the size of the structure and increase the number of
        dimensions by 1.
      Insert the new cofactors into the new structure
  Endif
Until No untested variables remain
```

Figure S1. Symmetry Detection.

```
Remove all State-Space transformations.
Check Back diagonals, stop if symmetry is detected.
Reverse Hyper Linear structure along one dimension
Check Back diagonals, stop if symmetry is detected.
Restore Hyper Linear Structure
Reverse Odd Rows
Check Back diagonals, stop if symmetry is detected.
Restore Hyper Linear Structure
Reverse Odd Columns
Check Back diagonals, stop if symmetry is detected.
Restore Hyper Linear Structure
Reverse Hyper Linear structure along one dimension
Reverse Odd Rows
Check Back diagonals, stop if symmetry is detected.
Restore Hyper Linear Structure
Reverse Hyper Linear structure along one dimension
Reverse Odd Columns
Check Back diagonals, stop if symmetry is detected.
```

Figure S2. General Symmetry Pair Detection.

```
For each plane containing the pair to be tested
  For each diagonal
    Comparator = HeadVertex.function;
    For V = each vertex after the head vertex
      If Comparator Not Equal V.function Then
        Report Failure
      Endif
    Endfor
  Endfor
Endfor
Report Success
```

Figure S3. Ordinary Symmetry Diagonal Check.

```
For each plane containing the pair to be tested
  For each diagonal
    Comparator = HeadVertex.function;
    AntiComparator = Negate(Comparator)
    Odd = 1;
    For V = each vertex after the head vertex
      If Odd = 1 Then
        If AntiComparator Not Equal V.function Then
          Report Failure
        Endif
        Odd = 0;
      Else
        If Comparator Not Equal V.function Then
          Report Failure
        Endif
        Odd = 1;
      EndIf
    Endfor
  Endfor
Endfor
Report Success
```

Figure S4. Anti-Symmetry Diagonal Check.

```
GateState[i] = GateState[i] + PortState;
PortState = -PortState;
// note the contents of the Then and Else sections
If Value[GateState] Not Equal OldGateState Then
  If EventQueued Then
    Dequeue Event
  Else
    Queue Event
  Endif
Else
  Do Nothing;
Endif
OldGateState = Value[GateState];
GoTo NextEvent
```

Figure S5. Ordinary Symmetry Event Processor.

```
GateState[i] = GateState[i] + PortState;
PortState = -PortState;
// note the contents of the Then and Else sections
If Value[GateState] Not Equal OldGateState Then
  Do Nothing;
Else
  If EventQueued Then
    Dequeue Event
  Else
    Queue Event
  Endif
Endif
OldGateState = Value[GateState];
GoTo NextEvent
```

Figure S6. Anti-Symmetry Event Processor.