Multilevel Logic Synthesis for Arithmetic Functions

Chien-Chung Tsai *

Mentor Graphics Corporation

Wilsonville, OR 97070-7777

Malgorzata Marek-Sadowska

Department of Electrical and Computer Engineering

University of California, Santa Barbara, CA 93106

Abstract -- The arithmetic functions, as a subclass of Boolean functions, have very compact descriptions in the AND and XOR operators. Any n-bit adder is a prime example. This paper presents a multilevel logic synthesis method which is particularly suited for arithmetic functions and utilizes their natural representations in the field GF(2). Algebraic factorization is performed to reduce the literal count. A direct translation of the AND/XOR representations of arithmetic functions into multilevel networks often results in excessive area, mainly due to the large area cost of XOR gates. We present a process of redundancy removal which reduces many XOR gates to single AND or OR gates without altering the functional behavior of the network. The redundancy removal process requires only to simulate a small and decidable set of primary input patterns. Preliminary results show that our method produces circuits, before and after technology mapping, with area improvement averaging 17% when compared to Berkeley SIS 1.2. The run time is reduced by at least 50%. The resulting circuits also have good testability and power consumption properties.

1. Introduction

The majority of the multilevel logic synthesis CAD tools currently on the market implement the methods based on techniques described in [2] [4]. The central theme of the synthesis methods is the factorization and decomposition of the original design described by macro blocks, each block being in the Sum-of-Product (SOP) form. Naturally, before the logic synthesis begins, it is necessary to determine a compact and concise description of the circuit's functional behavior which is suitable for the tool. Therefore, even the functions having their naturally compact descriptions in AND/XOR forms are changed into equivalent AND/OR forms and might be mixed into macro blocks with the rest of the function. This is also true for arithmetic functions [17], such as adders, multipliers, and error-correcting circuits that are originally derived in the context of algebraic field *GF(2)*. If input descriptions of arithmetic functions are represented in forms different from AND/XOR forms, e.g. two-level SOP forms, then the compact descriptions of the original equations are completely lost. In both cases, the synthesis tools rely entirely on the Boolean factorization to include XOR gates. This may result in synthesized circuits with suboptimal area. We will use two examples from the *IWLS'91* benchmark set [22] to illustrate our observations.

Example 1: t481 is a 16-input, single-output function and is listed in both the two-level and multilevel benchmark sets in [22]. The case name comes from the fact that there are *481* irredundant, prime cubes in the two-level SOP form. In the multilevel set, the user's guide lists *t481* as a circuit having *2072* gates (no information on the gate types). When we run the major scripts in the Berkeley SIS 1.2 to synthesize the function, the script *rugged* generated the best result that has only *237 2*-input AND/OR gates. The run time for the script is high with *1372* CPU seconds. In contrast, *t481* has only *16* cubes in the well-known Fixed-Polarity Reed-Muller (FPRM) form. The FPRM form is one of the many two-level AND\XOR representations [18]. Factorization methods from Elementary Algebra can be applied easily to generate a good multilevel circuit. After certain redundant XOR gates are replaced with AND/OR gates (we will explain the details later), the final result is a multilevel circuit described by the following equation.

$$t481 = \left(\overline{v_0}v_1 \oplus v_2\overline{v_3}\right)\left(\overline{v_4}v_5 \oplus \left(\overline{v_6} + v_7\right)\right) \oplus \left(\left(v_8 + \overline{v_9}\right) \oplus v_{10}\overline{v_{11}}\right)\left(\overline{v_{12}}v_{13} \oplus v_{14}\overline{v_{15}}\right).$$

It can be implemented by *25 2*-input AND/OR gates if each XOR gate is replaced by three AND/OR gates.

The adders have similar sub-optimal synthesis result when conventional tools are used. This is due to the same problem as illustrated in the next example.

Example 2: z4ml is a *3*-bit adder with a carry-in bit and a carry-out bit. Basic data [3] show that it has *59* irredundant, prime cubes in the two-level SOP form. In contrast, there are *32* cubes in the FPRM form. All the *32* cubes have a special property (we will explain this later) which facilitates the algebraic factorization nicely. We have generated a multilevel circuit for *z4ml* with *21 2*-input gates and the synthesis process does not rely on any high level description. The best result derived from SIS scripts has *24 2*-input gates. The run time for SIS is much higher than ours. The difference in size increases (in terms of percentage) for larger circuits as is the case of the *6*-bit adder *add6*.

In general, for arithmetic functions, the two-level SOP forms contain large set of cubes and the irredundant prime covers are difficult to derive and store [5] [9]. On the other hand, the FPRM forms of arithmetic functions have small sets of cubes [17]. The FPRM form of any function is canonical with fixed polarity of each variable, therefore the cubes in the FPRM form clearly indicate the relations among variables. Both of the above examples indicate these properties. For adders, e.g. *z4ml*, the FPRM forms are the same as the original equations, in two-level form specification. The transformation to SOP form will result in information loss. Therefore, maintaining the original equations in the FPRM forms for multilevel logic synthesis can shorten the synthesis time and assist in deriving better results. Note that we use the FPRM forms only as the initial specification.

The circuits implementing functions in FPRM forms, or any related forms, have been considered by only a small group of researchers besides Reed [15] and Muller [13]. Reddy [14] has shown that the circuits implementing FPRM forms have extremely good testability property. Recently, multilevel logic synthesis methods with XOR operators [19] or related forms [1] [11] [16] [17] [21] have achieved some good results, especially for Lookup Table based FPGAs. However, for standard cells, most of the current results assume that the XOR gate is a single entity. For arith-

* this work was conducted when the author was at the University of California, Santa Barbara.

33rd Design Automation Conference ®

DAC 96 - 06/96 Las Vegas, NV, USA

metic functions this will lead to additional cost, since each XOR gate has a relatively large area in comparison to AND/OR gates. Implementing an XOR gate by AND/OR gates is discussed in [1], but the process of redundancy removal is not addressed. All of the recently proposed multilevel synthesis methods with XOR gates use various types of decision diagrams with fixed variable ordering, but none of them apply complete algebraic factorization. This leads to suboptimal designs. Decision diagrams will not present automatically a good factorization and changing the variables' order does not solve the problem.

In this paper we propose a multilevel logic synthesis method for arithmetic functions which formalizes our observations described by the two examples above. The primary objective is to minimize the area. However, the resulting circuits also possess good testability properties and their estimated power dissipation is low. The test set for these circuits can be determined without conventional test generation methods [21]. The synthesis method is described in Sections 2 to 4 and contains the following steps:

(1) The FPRM form is generated, if the original specification is not in this form. We include this step in our experiment, since we do not have any benchmark cases described in FPRM form. All the cases in *IWLS'91* are in SOP forms. (Section 2)

(2) Algebraic factorization is performed to minimize the literal count and build the multilevel circuit. A set of rules to merge cubes is also discussed. We present two methods for algebraic factorization: the first method uses cubes directly and the second method uses decision diagrams to derive initial networks. (Section 3)

(3) A set of primary input patterns is generated from the cubes in the FPRM form and simulated. Based on the simulation results, some of the XOR gates can be reduced to single AND or OR gates without altering the functionality of the circuit. (Section 4)

Section 5 presents our experimental results. Conclusion and discussion of future improvements are in Section 6.

2. The FPRM Forms and their Functional Decision Diagrams

A FPRM form of a Boolean function is its representation as an XOR sum of cubes, in which every variable has either positive or negative (but not both) polarity in all the cubes. FPRM forms of a function can be efficiently derived and the cubes retrieved from the ordered functional decision diagram (OFDD) [12] [20] or directly from any two-level SOP form [20]. In our experiment, we use the OFDDs to derive the cubes for the first method of algebraic factorization. For the second method of algebraic factorization, the OFDDs are used to generate the initial multilevel networks. The OFDD can be derived efficiently [20] from reduced ordered binary decision diagram (ROBDD) [6]. In our implementation, we utilize the SIS 1.2 ROBDD package augmented by a polarity vector. The origination and structure of an OFDD is described as follows.

A FPRM form of a Boolean function is the expansion of each variable x_i with either the positive Davio expansion $f = x_i f^B_{x_i} \oplus f_{\overline{x}_i}$, or the negative Davio expansion $f = \overline{x}_i f^B_{x_i} \oplus f_{x_i}$ [8], where f_{x_i} and $f_{\overline{x}_i}$ are cofactors of f and $f^B_{x_i} = f_{x_i} \oplus f_{\overline{x}_i}$. For example, $f = \overline{x}_1 \oplus \overline{x}_1 x_3 \oplus \overline{x}_1 x_2 \oplus \overline{x}_1 x_2 x_3 \oplus x_3 \oplus x_2$ is the FPRM representation of f in polarity *(0 1 1)*.

With each n-input function f we associate a binary n-dimensional *polarity vector*. An entry of the vector is *0*(*1*) if the corresponding variable in the FPRM form is in the negative (positive) polarity. Note the binary nature of the Davio expansions. Each equation has two terms: one contains the literal t_i and the other does not. Using the construction of ROBDDs as an analogy, the variables are ordered and each variable is expanded by applying one of the Davio equations. The FPRM form of a function f can be expressed in a binary decision tree where terminal nodes *1* and *0* indicate the presence or absence of each path (cube) and the root of the graph represents *f*. In this binary tree, each level represents a variable and each node has a branch that contains the literal and a branch that does not. The OFDD is derived from the binary decision tree with all isomorphic subtrees merged and only two terminal nodes (*0* and *1)* present. When two subtrees directly under a nonterminal node are isomorphic, the node is eliminated and the subtrees are merged. The nonterminal nodes in our OFDD have branches labeled *0* and *1*. Agreement of the branch label with the polarity of the corresponding variable indicates the existence of the literal in the cube. The other branch, with a label different from the polarity, indicates the absence of the literal in the cube. Each path from the root to the terminal *one* node represents a set of cubes in *f*. Any missing node along a path corresponding to the variable x_j, represents two cubes in the FPRM form. One cube contains x_j with the appropriate polarity and the other cube does not have x_j. Therefore, a path with k nonterminal nodes stands for a set of 2^{n-k} cubes in the FPRM. For example, the OFDD in *Figure 1* represents $f = \overline{x}_1 \oplus \overline{x}_1 x_3 \oplus \overline{x}_1 x_2 \oplus \overline{x}_1 x_2 x_3 \oplus x_3 \oplus x_2$ with polarity vector *V = (0 1 1)*. In this OFDD, the path with $x_1 = 0$, represents the first four cubes and the path *101* represents the cube x_3. Note that the same OFDD can represent a different function if the polarity vector is different. Therefore, keeping a polarity vector with the OFDD is essential. We will refer to [1], [8], [12], [13], [15], [18] and [20] for more details on the FPRM forms and their OFDDs.

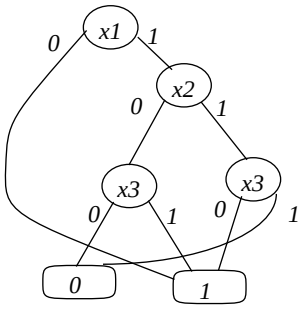

Figure 1 OFDD of f with V=(0 1 1)

In the sequel, we will refer the dependent variables of a function (cube) as the *support set*. Some cubes of the FPRM forms are called prime cubes. A cube *p* is *prime* [7] in the function *f* if the support set of *p* is not properly contained in any support set of the remaining cubes. Csanky *et al* [7] have proved that every prime cube occurs in all 2^n possible FPRM forms of a function. The set of prime cubes indicate sets of variables that are related. For example, *z4ml* is a three-bit adder that adds two binary numbers whose consecutive bits are $x_2x_3x_1$ and $x_5x_6x_4$ and the carry-in is x_7. The outputs are *x24, x25, x26, x27*, where *x24* is the carry-out bit. The output *x26* of *z4ml* in the FPRM form is $x26 = x_3 \oplus x_6 \oplus x_1x_4 \oplus x_1x_7 \oplus x_4x_7$, where all the cubes are prime. All the cubes in each output function of *z4ml* are primes. This can be very useful for algebraic factorization. The same property occurs in other arithmetic functions such as multipliers. In *t481, 10* of the *16* cubes are primes.

3. Algebraic Factorization

There are exactly *(m-1)* XOR operators in a FPRM form with *m* cubes. Therefore, the goal of factorization is to: (1) reduce as many XOR gates as possible by merging cubes, and (2) factor out the maximal number of variables from subsets of cubes. The multilevel network is constructed during factorization. We propose two factorization methods.

The first one is the cube method that takes all the cubes in the FPRM forms as the input. First the cubes are divided into groups

such that every two groups have disjoint supports. Each group is factored separately and the resulting subnetworks are joined by a balanced binary tree of XOR gates to form the complete network.

Let *A*, *B* and *C* be cubes or complex expressions and + represents the OR gate. The rules we apply to reduce the XOR gates are as follows.

The Reduction rules are: (a) $A \oplus AB = A\bar{B}$, (b) $AB \oplus AC \oplus ABC = A(B + C)$, (c) $AB \oplus \bar{B} = A + \bar{B}$.

The Factorization rules are:

(d) $AB \oplus AC \oplus A\ldots = A(B \oplus C \oplus \ldots)$,

(e) $AB + AC + A\ldots = A(B + C + \ldots)$.

Note that rule (e) is used only after Reduction rules have been applied. The two sets of rules are applied to the cubes iteratively until no further factorization is possible. The first factorization method is summarized in the following procedure:

Step 1: generate all the cubes in the FPRM form.

Step 2: divide the cubes into groups with disjoint support.

Step 3: for each group of cubes, divide the cubes into subgroups with maximal common support.

Step 4: apply Reduction or Factorization rules to each subgroup to generate subnetworks.

Step 5: merge subnetworks by a balanced, binary tree of XOR gates to form the complete network.

The second factorization method uses the OFDD to generate the initial, factored form. In the OFDD, any set of nodes that share a common child node represents a factored subexpression similar to the right hand side of the rule (d). The initial network is constructed by replacing each node of the OFDD with a set of one AND gate and one XOR gate that implement the appropriate Davio expansions. Note that, as described in the previous section, the variables that are missing in each path of the OFDD should be in some of the cubes; therefore, additional care is needed to include them. The initial network can be constructed by a single traversal of the OFDD. The following procedure summarizes the second factorization method:

Step 1: traverse the OFDD and construct the initial network.

Step 2: traverse the initial network and apply the Reduction and Factorization rules when possible.

We have implemented both of the factorization methods in our experiment and the results are comparable but the second method has better results on a few more test cases. For multioutput functions, we do factorization for each output function and use the SIS command *resub* to merge the networks of all output functions. Note that the missing variables in the paths of a multioutput OFDD could result in a node being shared by two output functions with different support variables; therefore, we can not use the multioutput OFDD to build the initial network directly.

In general, we believe that more elegant methods for algebraic factorization are still possible, similar to the methods in [2], for AND/XOR forms. The set of rules developed by Sasao [17] for XOR related forms could serve as a base and the main factorization technique could follow the methods in [2]. This new method should be targeting designs that have high level descriptions in macro blocks.

4. Redundancy Analysis of XOR Gates

The rules (a) to (c) of the previous section suggest that some of the XOR gates can be removed or reduced to simple OR gates. In this section we will show a complete method for detecting such reducible XOR gates in the entire network. For simplicity, in this section we make the following assumptions:

(1) all the variables have positive polarities in the FPRM forms. (Our results apply to FPRM forms with any polarity combinations)

(2) the cube *1* in the FPRM form, if exists, is always implemented as an inverter at the primary output (PO). This can be done since $f = \bar{f} \oplus 1$.

(3) the multilevel network, called N_x, has been constructed by applying the algebraic factorization described in Section 3, but reduction rules (a) - (c) were not applied.

Hayes [10] has proved that for a two-input XOR gate implemented by AND/OR gates, all four input patterns have to be applied to test internal single stuck-at (*s-a*) faults and this is independent of the implementation of the XOR gate. Therefore, the internal *s-a* faults can be partitioned into four classes of equivalent faults; each class is detected by a particular input pattern. A class of internal faults is untestable, if, and only if, the corresponding input pattern is either: (a) uncontrollable, or (b) unobservable. An *s-a-0* untestable fault means the wire is redundant and can be set to a constant *0*. Similarly, a *s-a-1* untestable wire can be set to a constant *1*. The gate with redundant inputs can then be simplified. For the XOR internal gates, if the whole class of faults corresponding to a particular input pattern is untestable, then all the wires corresponding to the faults can be set to constant values.

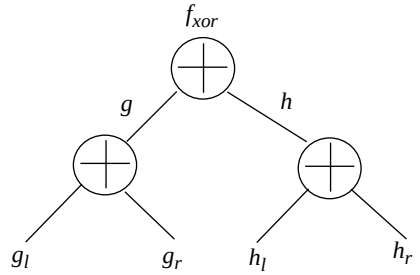

Figure 2 The structure of the XOR gates inside the network.

Let $f_{xor} = g \oplus h$ be an internal XOR gate of N_x, where *g* and *h* are the output functions of the fanins. Similarly, let $g = g_l \oplus g_r$ and $h = h_l \oplus h_r$. Figure 2 shows the structure of the consecutive XOR gates. Note that in the network built by the algebraic factorization, there might be AND gates in *g* or *h* that implement some subcube or subexpression; i.e., $g = A(g_l \oplus g_r)$ or $h = B(h_l \oplus h_r)$. However, we are concerned only with the consecutive XOR gates in the analysis; therefore, we do not show *A* (*B*) in the figure.

Property 1 If all primary inputs (PI) are set to *0*, then the inputs and output of every XOR gate are also *0*. This is true since we have the assumptions (1) and (2) above. We will denote the all zero PI pattern as AZ.

Property 2 For each two-input XOR gate in N_x, at least three of the four input patterns are controllable.

Proof: If less than three input patterns occur, than one of the inputs or the output is a constant. However, none of f_{xor}, *g*, or *h* should be constant, since each function is an XOR sum of a subset of cubes from the canonical FPRM form. (QED)

Property 1 guarantees that the *(0, 0)* input pattern is always controllable for any XOR gate in N_x. Now we consider the remaining three input patterns of any XOR gate in N_x. *Table 1* shows the respective functions.

Property 3 If the input pattern *(1, 1)* is uncontrollable or unob-

servable, then $f_{xor} = g + h$; i.e., the XOR gate is reduced to an OR gate.

Property 4 If the input pattern *(0, 1)* is uncontrollable or unobservable, then $f_{xor} = g\,\bar{h}$; i.e., the XOR gate is reduced to an AND gate. Similarly, if the input pattern *(1, 0)* is uncontrollable or unobservable, then $f_{xor} = \bar{g}\,h$.

Table 1 Truth Table for XOR and three implied functions

values at g and h	$g \oplus h$	$g + h$	$g\,\bar{h}$	$\bar{g}\,h$
0, 0	*0*	*0*	*0*	*0*
0, 1	*1*	*1*	*0*	*1*
1, 0	*1*	*1*	*1*	*0*
1, 1	*0*	*1*	*0*	*0*

The following property ensures that the reductions of the XOR gates can occur only in a certain order.

Property 5 A situation that an input pattern of f_{xor} is not observable occurs only when redundancy was discovered at some of the XOR gates in the transitive fanout of f_{xor}.

Proof: XOR gates do not have controlling value, therefore, observability is changed only when a path to any primary output contains some AND/OR gates. (QED)

When an XOR gate f_{xor} is reduced to an OR or AND gate, the observability at f_{xor} will be dominated by the controlling value. In the case where f_{xor} is changed to an OR gate, suppose that all the PI patterns that set both g_l and g_r to *1* also set *h* to *1*, then the *(1, 1)* input pattern of *g* is unobservable and the XOR gate *g* can be reduced. The following properties describe cases when redundancies occur due to observability problems.

Property 6 Suppose that XOR gate was reduced to $f_{xor} = g + h$. When for all PI patterns controlling g_l and g_r to *1*, *h* is also set to *1*, then $g_l = g_r = 1$ is unobservable and *g* is reduced to an OR gate. Similarly, when $g_l = 0$ and $g_r = 1$ implies $h = 1$, then $g = g_l\,\bar{g}_r$.

Property 7 Suppose that XOR gate has been reduced to $f_{xor} = g\,\bar{h}$. If all PI patterns controlling (g_l, g_r) *to (0, 1)* or *(1, 0)* also set *h* to *1*, then one of *(0, 1)* or *(1, 0)* is not observable and *g* can be reduced to an AND gate.

Similar properties can be derived for the case when the XOR gate is reduced to $f_{xor} = \bar{g}\,h$.

For multioutput functions, *Property 5* applies to all POs that share the f_{xor} subnetwork. If there exists a path from f_{xor} to any PO that contains only XOR gates, then the observability of the f_{xor} is still maintained. In this case Properties *6* and *7* do not apply.

The last three properties indicate that the observability redundancies are the consequences of controllability redundancies and they create a domino effect toward the PIs side.

Consider a single output function in its tree network N_x. For any XOR gate f_{xor} in N_x, as described above, the *AZ* pattern can set the inputs *g* and *h* of f_{xor} to *0*. To decide the controllability of the remaining three input patterns *(0, 1)*, *(1, 0)* and *(1, 1)*, we need an efficient method to determine all the values of *g* and *h* that can possibly occur. If *g* and *h* have totally disjoint support sets of PIs, then all input patterns at f_{xor} are controllable and observable and the XOR gate f_{xor} can never be reduced. For example, all the XOR gates in a parity function are not reducible. As described in the algebraic factorization, all the XOR gates in the balanced binary tree that form the POs are irreducible. Therefore, we will not check the redundancies of this type of XOR gates. In the remainder of this section, we will assume that the support sets of *g* and *h* have at least one PI in common.

We will construct a PI pattern set as follows: for each cube C_i in the FPRM form of the function, we create a PI pattern *Pi* where all the variables in C_i are set to *1* and all the variables not in C_i are set to *0*. We will call this set of PI patterns the one-cube (*OC*) set. All the PI patterns in *OC* are simulated on N_x. This will generate some input patterns of each XOR gate. However, we must guarantee that for each XOR gate all possible input patterns are determined; i.e., all possible PI combinations that could generate additional input patterns of f_{xor} have been considered.

Property 8 There exists at least one PI pattern in *OC* set that derives a *1* at an XOR gate f_{xor}.

Property 9 At least two of the three input patterns *(0, 1)*, *(1, 0)* and *(1, 1)* at each XOR gate are derived by patterns in the *OC* set.

For XOR gates that are in the direct fanout of two cubes, we know that all three input patterns are always controllable, since the one-cube PI pattern corresponding to each cube will derive *(1, 0)* and *(0, 1)* input patterns, respectively. For the *(1, 1)* input, we will simulate the *AO* pattern where all the variables are set to *1*.

After the simulation of *OC* and *AO* patterns, one of the three input patterns might not be derived at some f_{xor} in the network. Now the question is whether the missing input pattern is controllable. We know that based on the property of XOR operator, when an odd number of cubes in *g* are set to *1* and the rest of the cubes are set to *0* then *g* is *1*. If an even number of cubes in *g* are set to *1* then *g* is *0*. Therefore, the input values at f_{xor} are decided by the parity of the cubes that are set to *1* in *g* and *h*. Clearly we do not have to consider PI patterns that can not set any cube to *1* in N_x, since their effect on the inputs of each XOR gate is identical to the *AZ* pattern. However, there are still a large number of PI patterns that could derive various parities of cubes at *g* and *h*. To avoid enumerating all the possible PI patterns explicitly, we have derived a method that is based on the following strategies: (1) define an ordering of the cubes in the function and enumerate the parity value of cubes in order, and (2) maintain the record of only the accumulated parity values of *g* and *h* when at least one of them has parity *1*. Note that we do not have to simulate any PI pattern explicitly, since the parity values are enough to decide the functional values at *g* and *h*. The method is quite involved and we have to cut this portion due to the space limitation. For any XOR gate where one of the input pattern is missing, we apply the method to decide whether the missing input pattern is controllable. If a parity value combination of *g* and *h* is derived that matches the missing input pattern, then we generate a PI pattern by setting all the variables in all the related cubes to *1* and all other variables to *0*.

With complete information of all possible inputs to every XOR gate, the gate reduction is done as described in the following step.

1. Traverse the XOR gates starting from POs. For each XOR gate, if any of the three input patterns is missing, reduce the XOR gate to OR or AND based on either *Properties 3* or *Property 4*, respectively.

2. For each reduced gate, traverse backward toward PIs. For each XOR gate not yet reduced, if any condition of Properties *6* or *7* is satisfied, then reduce the XOR gate accordingly.

After redundancies are removed from XOR gates, the paths from some first level AND gate to the PO might contain some AND/OR gates. This could create redundancies in the fanins of these AND gates. When this occurs as untestable *s-a-1* fault, then the input can be set to the constant *1* and eliminated. If the redundancy occurs as *s-a-0* fault untestable, then the fanin can be set to the constant *0* and the AND gate output is also set to constant *0*. To verify the redundancy, for each input x_j in cube C_i, we generate a PI pattern directly from pattern P_i (in set *OC*) by switching the corresponding bit of x_j to *0*. For a *k*-variable cube, we will generate *k*

extra PI patterns. We will call this set *SA1* for testing *s-a-1* faults of the fanins on the first level AND gates. The PI patterns in *OC* are used for *s-a-0* faults.

As before, we simulate the pattern sets *OC* and *SA1*. The redundancy detection relies on the simulation results of the corresponding PI pattern for a particular fault. For each fanin of the first level AND gates, we check the simulation result along the path to the POs. If there exists an AND or OR gate on the path where the side input have the controlling value of the gate, then this gate will prevent any fault effect from propagating forward. Therefore, the corresponding fault is untestable. The particular fanin can then be set to constant and removed.

The following equalities show the order in which the redundancies are discovered and the corresponding reductions: $(B \oplus C) \oplus BC = (B \oplus C) + BC = (B + C) + BC = B + C$. Clearly, the Reduction rules (a) to (c) that are used in algebraic factorization can reduce the number of PI patterns needed for redundancy removal.

5. Experimental Results

We have implemented our method in C on a Sun Sparc 5 and have run the program on a set of *IWLS'91* benchmark circuits. The circuits derived from our program are compared with the original circuits by using the *verify* command in SIS 1.2. We compare our results with the best results of the three SIS scripts: *rugged, boolean* and *algebraic*. To make fair comparisons, we also run *red_removal* in SIS after the completion of the scripts to remove all redundant wires in circuits generated by SIS. In *Table 2*, the first column lists the names of the circuits (the known arithmetic circuits are in bold face) and the second column (*I/O*) lists the number of PIs and POs. The third and forth columns list the results before the technology mapping stage. In each column we show the number of literals of the circuits in 2-input AND/OR gates *(lits)* and the CPU time for the results *(time)*. The run time in our results includes the time to generate the OFDDs and to extract all cubes.

To realistically measure the results, we use the SIS *map* command for technology mapping. We use the cell library *mcnc.genlib* that has (1) *2*-input XOR/XNOR gates, (2) *2*-input AND/OR gates, (3) NAND/NOR gates of up to four inputs, and (4) four complex cells such as *AOI22*. Columns 5 and 6 of *Table 2* list the gate count *(gates)* and literal count *(lits)* of circuits after technology mapping. The column *improve%lits* shows the percentage improvement (or loss) of the mapped circuits when comparing our results with the SIS results.

As part of our on-going research, we have also run the power estimation for these circuits using SIS command *power_estimate* with default options. The percentage improvement (or loss) in power dissipation is listed in column *improve%power.*

The last two rows show the summation where the row *Total arith.* is the sum of all arithmetic circuits and the last row is the sum of all the circuits. The columns of improvements in these two rows are the average improvements.

The experimental results generally support our observations that XOR gates can be used to reduce circuit size in a more constructive way than the conventional synthesis methods do. Note that some of the circuits are not listed as arithmetic functions [22] and we do not know the functionality of these circuits.

6. Conclusions

In this paper we have proposed a method for multilevel logic synthesis which is suited for arithmetic functions or any functions with managable sizes of the FPRM forms. We have utilized the initial specifications of such arithmetic functions as adders and multipliers, by directly synthesizing the FPRM forms of Boolean functions. Algebraic factorization of the FPRM forms are much simpler then the factorization of SOP forms for arithmetic functions. Properties derived for the XOR gates are used to identify redundancies without test pattern generation. Our method is fundamentally different from conventional synthesis methods that are based on SOP forms. This leads to solutions in different search space and can be used to complement the weak points in the conventional synthesis methods. Our method is particularly useful for adders, multipliers, error checking circuits and functions related to coding theory. Preliminary results show a good percentage improvement as compared to SIS. We also note that our synthesis method produces irredundant networks with a complete test pattern set for single stuck-at faults.

The algebraic factorization and PI pattern set need further improvement to synthesize large, multioutput functions more efficiently. Other characteristics, such as power dissipation and delay, of the synthesized circuits will also differ from the results of conventional synthesis methods and need to be analyzed..

Acknowledgment: This work was supported in parts by NSF through grant MIP 9419119 and by California MICRO program.

7. References

[1] B. Becker and R. Drechsler, "Synthesis for testability: circuits derived from ordered Kronecker functional decision diagrams", *Technical Report, Universität Frankfurt, 14/94*, Fachbereich Informatik, 1994.

[2] R. K. Brayton and C. McMullen, "The decomposition and factorization of Boolean expressions", *Proc. IEEE International Symposium on Circuits and Systems*, pp. 49-54, May 1982.

[3] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[4] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A. Wang, "MIS: A multiple-level logic optimization system", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, pp. 1062-1081, Nov. 1987.

[5] Y. Breitbart and K. Vairavan, "The computational complexity of a class of minimization algorithms for switching functions", *IEEE Trans. Computers*, vol. C-28, pp. 941-943, Dec. 1979.

[6] R. E. Bryant, "Graph-based algorithms for Boolean functions manipulation", *IEEE Trans. Computers*, vol. C-35, pp. 677-691, Aug. 1986.

[7] L. Csanky, M. Perkowski and I. Schaefer, "Canonical restricted mixed-polarity exclusive-OR sums of products and the efficient algorithm for their minimisation", *IEE Proceedings-E*, Vol. 140, No. 1, pp. 69-77, Jan. 1993.

[8] M. Davio, J. P. Deschamps and A. Thayse, *Discrete and Switching Functions*, McGraw-Hill International, 1978.

[9] S. Devadas, K. Keutzer and S. Malik, "A synthesis-based test generation and compaction algorithm for multifaults", *Proc. 28st ACM/IEEE Design Automation Conference*, pp. 359-365, June 1991.

[10] J. P. Hayes, "On realizations of Boolean functions requiring a minimal or near-minimal number of tests", *IEEE Trans. Comp.*, vol. C-20, pp. 1506-1513, Dec. 1971.

[11] U. Kebschull, E. Schubert and W. Rosenstiel, "Multilevel logic synthesis based on functional decision diagrams", *Proc. European Design Automation Conf. '92*, pp. 43-47, Feb. 1992.

[12] U. Kebschull and W. Rosenstiel, "Efficient graph-based computation and manipulation of functional decision diagrams", *Proc. European Design Automation Conf. '93*, pp. 278-282, Feb. 1993.

[13] D. E. Muller, "Application of Boolean algebra to switching circuit design and to error detection", *IRE Trans. Electronic Computers*, EC-3, pp. 6-12, 1954.

[14] S. M. Reddy, "Easily testable realizations for logic functions", *IEEE Trans. Comp.*, vol. C-21, pp. 1183-1188, Nov. 1972.

[15] I. S. Reed, "A class of multiple-error-correcting codes and the decoding scheme", *IRE Trans. Information Theory PGIT-4*, pp.

38-49, 1954.

[16] A. Sarabi, P. F. Ho, K. Iravani, W. R. Daasch and M. Perkowski, "Minimal multi-level realization of switching functions based on Kronecker functional decision diagrams", *Proc. Intl. Workshop on Logic Synthesis '93*, P3a, 1993.

[17] T. Sasao, "Logic synthesis with EXOR-gates", *in Sasao, editor: Logic Synthesis and Optimization*, Kluwer Academic Publishers, pp. 259-285, 1993.

[18] T. Sasao, "AND-EXOR expressions and their optimization", *in Sasao, editor: Logic Synthesis and Optimization*, Kluwer Academic Publishers, pp. 287-312, 1993.

[19] B. Steinbach and A. Wereszczynski, "Synthesis of multi-level circuits using EXOR-gates", *Proc. IFIP WG 10.5 Workshop on Applications of Reed-Muller Expansion in Circuit Design*, Japan, pp. 161-168, August 1995.

[20] C. Tsai and M. Marek-Sadowska, "Minimisation of fixed-polarity AND/XOR canonical networks", *IEE Proc.*, vol. 141, Pt. E, No. 6, pp. 369-374, Nov. 1994.

[21] C. Tsai and M. Marek-Sadowska, "Logic synthesis for testability", accepted to *The Sixth Great Lakes Symposium on VLSI 1996*.

[22] S. Yang, "Logic synthesis and optimization benchmarks user guide—version 3.0", Microelectronics Center of North Carolina, Jan. 1991.

Table 2 Test results before and after Technology Mapping

		SIS		Ours		SIS		Ours		improve	improve
Circuit	I/O	lits	time	lits	time	gates	lits	gates	lits	%lits	%power
5xp1	**7/10**	**213**	**6.7**	**181**	**5.21**	**78**	**207**	**66**	**161**	**22**	**16**
9sym	**9/1**	**414**	**14.5**	**156**	**2.45**	**139**	**372**	**64**	**146**	**61**	**57**
adr4	**8/5**	**62**	**1.8**	**48**	**0.45**	**28**	**59**	**23**	**48**	**19**	**31**
add6	**12/7**	**114**	**3.2**	**76**	**0.91**	**48**	**106**	**44**	**82**	**23**	**42**
addm4	**9/8**	**700**	**465.0**	**588**	**42.22**	**221**	**573**	**224**	**539**	**6**	**13**
bcd-div3	**4/4**	**52**	**0.9**	**52**	**0.43**	**20**	**51**	**22**	**54**	**-6**	**-1**
cc	21/20	84	2.8	84	2.68	44	89	42	88	1	3
co14	**14/1**	**128**	**5.8**	**88**	**2.73**	**50**	**118**	**50**	**98**	**17**	**14**
cm163a	16/5	74	2.2	66	1.33	28	65	30	68	-5	13
cm82a	5/3	34	0.6	28	0.5	14	31	16	32	-3	29
cm85a	11/3	80	1.7	84	1.48	33	77	41	84	-9	1
cmb	16/4	86	2.2	37	0.22	32	83	17	50	40	35
f2	**4/4**	**36**	**1.2**	**34**	**0.73**	**16**	**40**	**16**	**34**	**15**	**12**
f51m	**8/8**	**187**	**8.6**	**137**	**2.71**	**66**	**160**	**63**	**132**	**17**	**27**
frg1	28/3	183	7.9	146	56.8	82	192	57	141	27	44
i1	25/13	70	2.1	61	1.9	33	73	34	69	5	3
i3	132/6	252	7.7	260	8.41	58	184	90	224	-22	24
i4	192/6	436	13.9	448	67.9	114	380	145	384	-1	7
i5	133/66	264	9.5	264	28.33	165	330	165	330	0	0
m181	**15/9**	**148**	**5.1**	**148**	**5.17**	**54**	**144**	**56**	**162**	**-13**	**-4**
majority	5/1	18	0.4	16	0.21	8	17	7	16	6	14
misg	56/23	138	4.4	100	6.11	52	132	41	95	28	27
mish	94/34	180	4.6	143	2.31	63	153	64	157	-3	0
mlp4	**8/8**	**534**	**19.3**	**452**	**12.72**	**176**	**503**	**171**	**411**	**18**	**21**
my_adder	**33/17**	**336**	**6.9**	**224**	**13.04**	**111**	**290**	**113**	**226**	**22**	**38**
parity	**16/1**	**90**	**1.2**	**90**	**0.28**	**15**	**60**	**15**	**60**	**0**	**0**
pcle	19/9	110	2.5	96	2.09	50	121	44	92	24	26
pcler8	27/17	156	4.8	135	5.12	73	153	73	137	10	4
pm1	16/13	69	2.8	65	1.44	33	67	39	73	-9	2
radd	**8/5**	**64**	**2.7**	**48**	**0.41**	**26**	**58**	**25**	**52**	**10**	**41**
rd53	**5/3**	**52**	**2.0**	**50**	**0.33**	**24**	**53**	**25**	**50**	**6**	**0**
rd73	**7/3**	**108**	**9.3**	**90**	**0.87**	**46**	**103**	**41**	**88**	**15**	**9**
rd84	**8/4**	**256**	**97.2**	**138**	**1.11**	**83**	**225**	**66**	**137**	**39**	**38**
shift	19/16	398	6.6	306	16.36	114	313	86	307	2	-8
sqr6	**6/12**	**212**	**4.2**	**217**	**4.05**	**72**	**194**	**82**	**223**	**-15**	**1**
squar5	**5/8**	**92**	**2.7**	**104**	**0.90**	**37**	**92**	**46**	**104**	**-13**	**5**
sym10	**10/1**	**430**	**711.1**	**176**	**4.53**	**133**	**350**	**78**	**179**	**49**	**59**
t481	**16/1**	**474**	**1372.4**	**50**	**0.69**	**190**	**438**	**23**	**48**	**89**	**85**
tcon	17/16	48	1.3	48	0.28	17	73	17	73	0	0
xor10	**10/1**	**54**	**1692.1**	**54**	**0.56**	**9**	**36**	**9**	**36**	**0**	**0**
z4ml	**7/4**	**48**	**1.7**	**42**	**1.05**	**25**	**50**	**21**	**42**	**16**	**11**
Total arith		4804	4435.6	3243	103.55	1667	4282	1343	3112	17.3	22.4
Total all		7484	4513.6	5630	307.02	2680	6815	2351	5532	11.9	18.0