On Synthesis-for-Testability of Combinational Logic Circuits

Irith Pomeranz and Sudhakar M. Reddy +
Electrical and Computer Engineering Department
University of Iowa
Iowa City, IA 52242

Abstract

We propose a synthesis method that modifies a given circuit to reduce the number of gates and the number of paths in the circuit. The synthesis procedure is based on replacing subcircuits of the given circuit by structures called comparison units. Comparison units are fully testable for stuck-at faults and for path delay faults. In addition, they have small numbers of paths and gates. These properties make them effective building blocks for synthesis of testable circuits. Experimental results demonstrate reductions in the number of gates and paths and increased path delay fault testability. The random pattern testability for stuck-at faults remains unchanged.

1. Introduction

In this work, we define a special class of functions, called *comparison functions*. Informally stated, a comparison function is a function that can be specified by providing a permutation X of its inputs and two bounds, L and U. Under the permutation X, every minterm where the function assumes the value 1 has a decimal value between L and U. Comparison functions have the property that they can be implemented by circuits referred to here as *comparison unit*s, which are efficient in terms of the number of gates they require, have a small number of paths going through them, and all the path delay faults in them are robustly testable. Hence the usefulness of comparison functions in synthesizing area efficient circuits which are highly testable.

Comparison functions are utilized in this work to reduce the gate count and the path count of combinational circuits through local circuit modifications. Subcircuits realizing comparison functions are identified and replaced by comparison units whenever such a replacement reduces the number of paths or the number of gates in the circuit. Area reduction by local circuit modifications was considered in [1-3]. Local modifications to enhance testability for path delay faults were considered in [4-6]. The advantage of comparison functions in this respect is that they provide a simple and uniform method of selecting which subcircuits should be modified and what their new structure is. In addition, significant reductions in the number of paths are achieved, which are larger than any reductions reported elsewhere. Experimental results demonstrate that the reduction in the number of paths results mainly in reducing the number of untestable path delay faults. Thus, the path delay fault testability is enhanced. In addition, the random pattern testability of the circuit does not deteriorate.

+ Research supported in part by NSF Grant No. MIP-9220549, and in part by NSF Grant No. MIP-9357581

32nd ACM/IEEE Design Automation Conference ®

The motivation for the reduction of gate count or area is straightforward. Next, we consider the motivation for reducing the number of paths. The path delay fault model was proposed to model defects that change the timing behavior of a circuit [7]. It is the most general of all delay fault models, since it models distributed as well as localized excessive delays. However, three problems are associated with this fault model, that prevent test generation procedures from achieving complete or close-to-complete fault coverage. (1) The number of paths (and therefore the number of path delay faults) in practical circuits may be very large [8]. (2) The number of tests to detect all path delay faults may be very large [5]. (3) Many path delay faults in practical circuits are not testable [9].

The problem of handling large numbers of paths was alleviated in part by the non-enumerative methods of [8,10], however, even using these techniques, the fault coverage for large circuits is very low. This is due in part to the large number of tests to detect all faults, and in part to the fact that many of the faults are untestable. In [11,12] it was shown that some path delay faults do not have to be tested, as correct speed of operation can be guaranteed by testing other faults. However, even when using this approach, the fault coverage obtained is sometimes low. In [13], a test-point insertion method to increase the testability of a circuit to path delay faults was presented. However, test-point insertion has the disadvantages of area and test application overheads, especially if a large number of test-points is needed to achieve the desired fault coverage. All the methods above, as well as test generation and fault simulation methods, can benefit from a reduction in the number of paths in the circuit under consideration. This is achieved by the proposed circuit modification procedure based on comparison functions. Moreover, experimental results show that a significant reduction in the number of untestable paths is achieved, whereas the number of testable paths increases. Thus, the fault coverage of path delay faults is significantly increased. The random pattern testability for stuck-at faults remains unchanged. This is important since resynthesis could cause the random pattern testability to deteriorate [16].

The paper is organized as follows. The required background is presented in Section 2. Comparison functions, circuits to implement them and their relationship to threshold functions are described in Section 3. In Section 4 we describe resynthesis procedures based on comparison functions. Experimental results are presented in Section 5. Section 6 concludes the paper.

2. Preliminaries

To compute the number of paths in a given circuit, we use the following procedure [8]. The procedure attaches a label $N_p(g)$ to each line g, equal to the number of paths from the primary inputs to line g. The procedure starts by assigning to every primary input the label 1. It then proceeds from inputs to outputs. The

output of a gate is labeled by the sum of its input labels. This is because for every path from a primary input to any of the gate inputs there is also a path from the same primary input to the gate output, and no other paths to the gate output exist. A fanout branch is labeled by the same label assigned to its stem. The total number of paths is equal to the sum of the primary output labels. The complete procedure is given next. We assume that the number of lines in the circuit is N_L, and that they are indexed $1,2, \cdots, N_L$ in *BFS* order from inputs to outputs.

Procedure 1: Computing the number of paths

(1) Assign all the primary inputs the label $N_p = 1$. Set $g = 1$

(2) If line g is the output of a gate with inputs $\{g_i: 1 \le i \le k\}$ labeled $\{N_p(g_i): 1 \le i \le k\}$, then label line g by $\sum_{i=1}^{k} N_p(g_i)$.

(3) If line g is a fanout branch of a stem g' labeled $N_p(g')$, label line g by $N_p(g')$.

(4) Set $g = g+1$. If $g \le N_L$, go to Step 2.

(5) The total number of paths is $\sum\{N_p(i)$: *i is a primary output*$\}$.

One of the goals of this work is to reduce the number of paths in a circuit by performing local modifications. To demonstrate the effect that such modifications can have on the number of paths, consider a k-input single-output subcircuit C' with inputs $\{g_i: 1 \le i \le k\}$ and output g. Suppose that inside C', there are $K_p(g_i)$ paths from g_i to g. Then the number of paths from the primary inputs to g can be expressed as $N_p(g) = \sum_{i=1}^{k} N_p(g_i) \cdot K_p(g_i)$. If we change the structure of C' (without changing the rest of the circuit) such that smaller values of $K_p(g_i)$ would correspond to larger values of $N_p(g_i)$, then we reduce the number of paths to g, and consequently, the total number of paths in the circuit. The following example demonstrates this point.

Example: Consider the 4-input, single-output function f_1 that has the following two (equivalent) minimal sum-of-products expressions.

$$f_{1,1} = \bar{x}_1 x_2 x_4 + x_1 \bar{x}_2 \bar{x}_3 + x_2 \bar{x}_3 x_4$$
$$f_{1,2} = \bar{x}_1 x_2 x_4 + x_1 \bar{x}_2 \bar{x}_3 + x_1 \bar{x}_3 x_4.$$

$K_p(x_i)$ is equal to the number of times x_i or $\bar{x}_i$ appear in the expression for f_1. Thus, for the first implementation $f_{1,1}$, $K_{p,1}(x_1) = 2$, $K_{p,1}(x_2) = 3$, $K_{p,1}(x_3) = 2$ and $K_{p,1}(x_4) = 2$. For the second implementation $f_{1,2}$ we have $K_{p,2}(x_1) = 3$, $K_{p,2}(x_2) = 2$, $K_{p,2}(x_3) = 2$ and $K_{p,2}(x_4) = 2$. Suppose that f_1 is implemented by a subcircuit C' embedded in a circuit C. Let $N_p(x_1) = 10$, $N_p(x_2) = 100$, $N_p(x_3) = 20$ and $N_p(x_4) = 20$. Since $K_{p,1}(x_1) < K_{p,2}(x_1)$, $K_{p,1}(x_2) > K_{p,2}(x_2)$, $K_{p,1}(x_3) = K_{p,2}(x_3)$, $K_{p,1}(x_4) = K_{p,2}(x_4)$ and $N_p(x_1) < N_p(x_2)$, the second implementation will result in a smaller number of paths. Specifically, under the first implementation, $N_p(f_{1,1}) = 2 \cdot 10 + 3 \cdot 100 + 2 \cdot 20 + 2 \cdot 20 = 400$, and under the second implementation $N_p(f_{1,2}) = 3 \cdot 10 + 2 \cdot 100 + 2 \cdot 20 + 2 \cdot 20 = 310$. □

The problem of reducing the number of gates (paths) by local circuit modifications is the following. Given a circuit C, find a set of subcircuits $C'_1, C'_1, \cdots, C'_k$ of C and a set of subcircuits $D'_1, D'_1, \cdots, D'_k$, such that if C'_i is replaced by D'_i, $1 \le i \le k$, then the function implemented by the circuit C does not change and the number of gates (paths) in C is minimized.

To simplify the problem, we impose the following constraints.

(1) We require that C'_i and D'_i would implement the same function. This ensures that the subcircuits can be selected independently.

(2) We set an upper limit on the number of inputs to a subcircuit C'_i that would be considered for replacement. This restricts the complexity of the search for C'_i and its replacement D'_i.

(3) We restrict the structure of the replacement subcircuit D'_i, as explained below. This reduces the complexity of the search for D'_i.

In selecting the structure of D'_i, our main objective is to obtain a structure that is fully testable, and has a small number of gates and a small number of paths through it. The structure is introduced in the following section.

3. Comparison functions

In this section, we introduce the class of *comparison functions*. We consider their implementation by a *comparison unit* that has at most two paths from any one of its inputs to its output. We also consider special cases where the number of paths from certain inputs of a comparison unit to its output is lower than two. We demonstrate that comparison units are fully robust-testable for path delay faults. Finally, we consider the identification of comparison functions.

3.1 Definition and implementation

Comparison functions are defined as follows.

Definition 1: Let $(x_1, x_2, \cdots, x_n)$ be a permutation of the input variables $\{y_1, y_2, \cdots, y_n\}$ of a function $f(y_1, y_2, \cdots, y_n)$. Let the minterms where $f(x_1, x_2, \cdots, x_n) = 1$ be $M = \{m_1, m_2, \cdots, m_k\}$, where m_i is given in decimal form (e.g., the minterm 00110 of a 5-input function has the decimal value 6). Then the function $f(y_1, y_2, \cdots, y_n)$ is said to be a *comparison function* if there exists a permutation $(x_1, x_2, \cdots, x_n)$ and two integers L and U, such that $m \in M$ if and only if $L \le m \le U$. □

The basic building blocks for implementing a comparison function are referred to as *comparison blocks*. There are two types of comparison blocks, the $\ge L$ block and the $\le U$ block. A $\ge L$ block produces the output 1 when supplied with an input combination whose decimal value is larger than or equal to L. A $\le U$ block produces the output 1 when supplied with an input combination whose decimal value is smaller than or equal to U. The implementation of $\ge L$ and $\le U$ blocks is considered below. The following example shows how these blocks can be used to implement a comparison function using a structure called a comparison unit. Note that we use x_1 as the most significant bit and x_n as the least significant bit.

Example: Consider the 4-input, single-output function $f_2(y_1, y_2, y_3, y_4)$ which is 1 for minterms {0001, 0101, 0110, 1001, 1010, 1110} or in decimal form, {1, 5, 6, 9, 10, 14}. Consider the permutation (x_1, x_2, x_3, x_4), where $x_1 = y_4$, $x_2 = y_3$, $x_3 = y_2$ and $x_4 = y_1$. The function $f(x_1, x_2, x_3, x_4)$ is 1 for the minterms {0101,0110,0111,1000,1001,1010}. In decimal form we have {5,6,7,8,9,10}, i.e., all the minterms between 5 and 10 (inclusive). The function can be implemented by the structure shown in Figure 1. In this implementation, $f_2 = 1$ if and only if an input combination m is applied, such that $5 \le m \le 10$. In this case, $L = 5 = (0101)$ and $U = 10 = (1010)$. □

The structure shown in Figure 1 is referred to as a *comparison unit*. An implementation of a $\ge L$ block is shown in Figure 2(a) for $L = (l_1 l_2 \cdots l_n)$. The gate types in Figure 2(a) are determined as follows.

$$G_i = \begin{cases} AND & \text{if } l_i = 1 \\ OR & \text{if } l_i = 0 \end{cases}$$

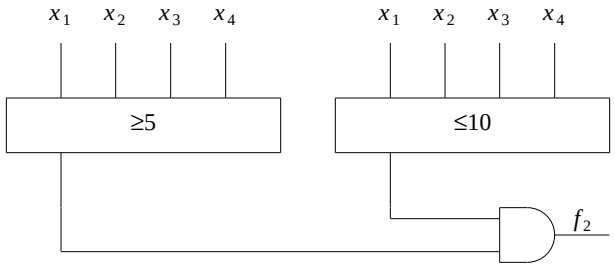
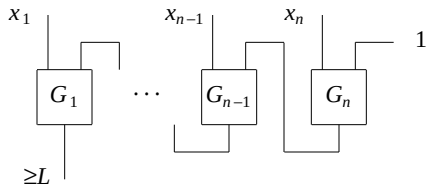
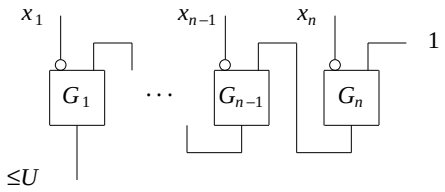


Figure 1: The implementation of f_2

The gate G_n is replaced by a direct connection to x_n when $l_n = 1$, and replaced by the constant 1 when $l_n = 0$. In both cases, G_n is omitted. Additional gates may then be omitted. For illustration, the implementation of a $\geq L$ block is shown in Figure 3(a) for $L = 3 = (0011)$. It can be seen that if $x_1 = 1$ or $x_2 = 1$, then the input combination represents a number larger than 3 and the output is 1, as required. If $x_1 = x_2 = 0$, then the output is 1 only if $x_3 = x_4 = 1$. The implementation of a $\geq L$ block for $L = 12 = (1100)$ is shown in Figure 3(b). This block demonstrates how the rightmost gates are omitted when the lower bound ends with 0s. Any input combination where $x_1 = x_2 = 1$ is larger than or equal to 12 and produces a 1 output.



(a) The $\geq L$ block



(b) The $\leq U$ block

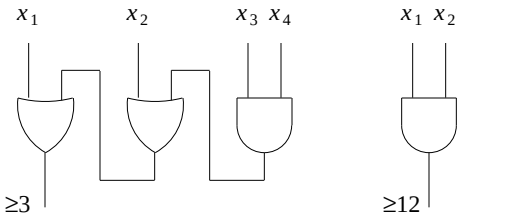
Figure 2: Implementations of the $\geq L$ and $\leq U$ blocks

An implementation of a $\leq U$ block is shown in Figure 2(b) for $U = (u_1 u_2 \cdots u_n)$. The gate types in Figure 2(b) are determined as follows.

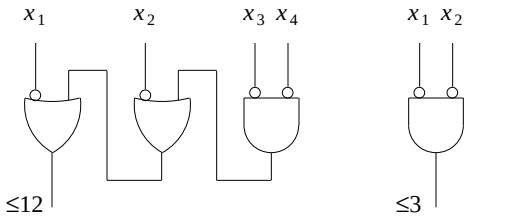
$$G_i = \begin{cases} AND & \text{if } u_i = 0 \\ OR & \text{if } u_i = 1 \end{cases}$$

The gate G_n is replaced by an inverter driven by x_n when $u_n = 0$, and replaced by the constant 1 when $u_n = 1$. Additional gates may then be omitted. For illustration, the implementation of a $\leq U$ block is shown in Figure 3(c) for $U = 12 = (1100)$. It can be seen that if $x_1 = 0$ or $x_2 = 0$, then the output value is 1, as required. If $x_1 = x_2 = 1$, then the output is 1 only if $x_3 = x_4 = 0$. The implementation of a $\leq U$ block for $U = 3 = (0011)$ is shown in Figure 3(d). This block demonstrates how the rightmost gates are omitted when the lower bound ends with 1s.

Next, we consider the number of paths through the comparison blocks. From Figure 2 it can be seen that there is at most one path from an input x_i to the output of a comparison block.



(a) The ≥3 block **(b) The ≥12 block**



(c) The ≤12 block **(d) The ≤3 block**

Figure 3: Implementations of the ≥3, ≥12, ≤12 and ≤3 blocks

Comparison units have the structure shown in Figure 1. In a comparison unit there are at most two paths from any input to the output of the unit. Thus, a comparison unit can be used to reduce the number of paths for subcircuits that have larger numbers of paths through them. Depending on L and U, an input may be omitted altogether from the corresponding comparison block, as shown in Figure 3(b,d). Thus, there may be only one path, or no paths at all from an input of a comparison unit to its output. Additional special cases exist, where the number of paths for some of the inputs is lower than two. These cases are considered in the following subsection.

The longest path through a comparison block has at most n two-input gates, where n is the number of inputs of the comparison function. Special cases exist as demonstrated in Figures 3(b,d) and as discussed in Subsection 3.2 below. In addition, when k consecutive gates have the same type, they can be combined into a $k+1$ input gate. For example, a ≥7 comparison unit is shown in Figure 4, where the two rightmost AND gates are combined into a single three-input gate. During the synthesis process, the number of logic levels can be kept low by considering comparison functions with small numbers of inputs. Experimental results presented in Section 5 show that the resynthesized circuits have similar delays to the original circuits.

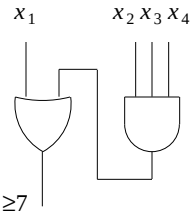


Figure 4: A comparison block for $L = 7$.

It is possible to implement any given function f as $f = f_1 + f_2 + \cdots + f_k$, where f_i is a comparison function for every $1 \leq i \leq k$. This can be accomplished by partitioning the set of minterms of f into subsets, each covered by a different function f_i. The function f can then be implemented using k comparison units driving an OR gate. In this work, we consider only com-

parison functions, that can be implemented by a single comparison unit.

Finally, we consider the relationship between comparison functions and threshold functions [14]. Consider a comparison function with input permutation $(x_1, x_2, \cdots, x_n)$ and bounds L and U. The $\geq L$ comparison block can be implemented by a threshold function with a weight 2^{n-i} assigned to x_i and with a threshold value $T = L$. This function would produce an output 1 iff the corresponding input combination is larger than or equal to L (i.e., $\sum_{i=1}^{n} x_i 2^{n-i} \geq L$), as required. Instead of the $\leq U$ block it is possible to use a $\geq U+1$ block implemented by a threshold function with the same weights as above and with $T = U+1$. The complemented output of the $\geq U+1$ block is then equivalent to the output of a $\leq U$ block. This output and the output of the $\geq L$ block can be ANDed to obtain the required function.

3.2 Special cases

In this section, we consider several special cases that simplify the comparison units.

3.2.1 Free variables

Let $f(x_1, x_2, \cdots, x_n)$ be a comparison function with a lower bound $L = (l_1 l_2 \cdots l_n)$ and an upper bound $U = (u_1 u_2 \cdots u_n)$. We define the set of *free variables* as follows.

Definition 2: $X_F = \{x_1, x_2, \cdots, x_F\}$ is a set of free variables if $l_i = u_i$ for every $1 \leq i \leq F$.

For example, consider a four-input comparison function with $L = 5 = (0101)$ and $U = 7 = (0111)$. Then $X_F = \{x_1, x_2\}$. The values of the free variables of a function f are the same for every minterm that sets f to 1. As a result, f can be implemented using the structure shown in Figure 5. The $\geq L_F$ and the $\leq U_F$ blocks have $n-F$ inputs, with $L_F = (l_{F+1} l_{F+2} \cdots l_n)$ and $U_F = (u_{F+1} u_{F+2} \cdots u_n)$. The F free variables drive the output AND gate directly (if their value is 1 in L and U) or through an inverter (if their value in L and U is 0). The number of paths from a free variable to the output of a comparison unit is one.

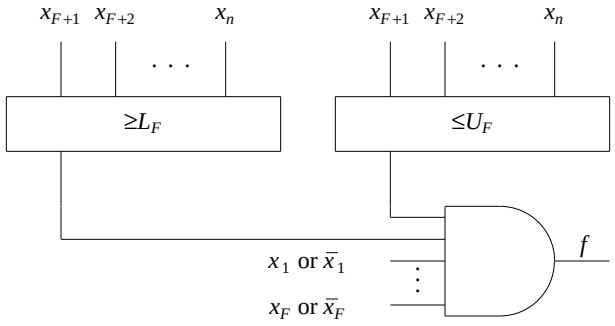


Figure 5: A function with free variables

3.2.2 Trivial lower or upper bounds

Let $f(x_1, x_2, \cdots, x_n)$ be a comparison function with a lower bound $L = (l_1 l_2 \cdots l_n)$ and an upper bound $U = (u_1 u_2 \cdots u_n)$. Let $X_F = \{x_1, x_2, \cdots, x_F\}$ be free variables. Let m_F be the minterm m restricted to the non-free variables, i.e., m_F is defined over variables $x_{F+1}, \cdots, x_n$.

Suppose that $L_F = (l_{F+1} l_{F+2} \cdots l_n) = (00 \cdots 0)$. Then any minterm m_F is larger than or equal to the lower bound L_F. In this case, the $\geq L_F$ block can be omitted. The number of paths from every input of the comparison unit to its output is at most one in this case.

Suppose that $U_F = (u_{F+1} u_{F+2} \cdots u_n) = (11 \cdots 1)$. Then any minterm m_F is smaller than or equal to the upper bound U_F. In this case, the $\leq U_F$ block can be omitted. The number of paths from every input of the comparison unit to its output is at most one in this case.

If $L_F = (00 \cdots 0)$ and $U_F = (11 \cdots 1)$, then the function f can be implemented by a single AND gate driven by the free variables. This case occurs when f has a single prime implicant. For example, if $f(y_1 y_2 y_3) = y_1 y_3$, then we use the permutation $x_1 = y_1$, $x_2 = y_3$ and $x_3 = y_2$. Under this permutation, $L = 6$ and $U = 7$. We obtain the set of free variables $X_F = \{x_1, x_2\}$, with $L_F = (0)$ and $L_U = (1)$. Every minterm m_F is between 0 and 1, therefore, the function is implemented by a single AND gate driven by x_1 and x_2 (or y_1 and y_3).

3.3 Testability of comparison units

In this subsection, we demonstrate that comparison units implemented according to Figure 5 (i.e., where the free variables drive the output AND gate) are fully robustly testable for path delay faults. The complete proof is omitted for space considerations. Let f be a comparison function with inputs $\{x_1, x_2, \cdots, x_n\}$, $L = (l_1 l_2 \cdots l_n)$ and $U = (u_1 u_2 \cdots u_n)$, and let the set of free variables be $X_F = \{x_1, x_2, \cdots, x_F\}$. To describe two-pattern tests for path delay faults, we use the values 000 and 111 to denote stable 0 and 1 values, respectively, and we use $0x1$ and $1x0$ to denote rising transitions and falling transitions, respectively. We demonstrate the construction of a complete test set in the following example.

Example: Consider the comparison unit shown in Figure 6. In this case, $L = 11$, $U = 12$, $X_F = \{x_1\}$, $L_F = 3$ and $U_F = 4$.

To test the path delay faults starting from x_1, we set $x_1 = 1x0$ or $x_1 = 0x1$ (depending on the fault). To set the other two inputs of the AND gate to 111, we apply to $(x_2 x_3 x_4)$ an input combination between 3 and 4. In this example, we apply 3, or (000,111,111).

To test the path delay faults starting from x_2 and going through the $\geq L_F$ block, we set $x_2 = 0x1$ or $x_2 = 1x0$. To propagate the fault, we set $x_3 = x_4 = 000$. This corresponds to the smallest possible decimal value that propagates the transition on x_2 to the output and it results in the output of the $\geq U_F$ block being 111 (larger decimal values may not yield 111 on the output of the $\geq U_F$ block). In addition, we set $x_1 = 111$.

To test the path delay faults starting from x_3 and going through the $\geq L_F$ block, we set $x_3 = 0x1$ or $x_3 = 1x0$. To propagate the fault, we set $x_2 = 000$ and $x_4 = 111$. This results in the output of the $\geq U_F$ unit being 111. In addition, we set $x_1 = 111$.

The remaining tests are derived in a similar way. The complete test set is shown in Table 1. □

Table 1: An example of a test set

fault	x_1	x_2	x_3	x_4
x_1	$0x1, 1x0$	000	111	111
$x_2, \geq L_F$	111	$0x1, 1x0$	000	000
$x_3, \geq L_F$	111	000	$0x1, 1x0$	111
$x_4, \geq L_F$	111	000	111	$0x1, 1x0$
$x_2, \leq U_F$	111	$0x1, 1x0$	111	111
$x_3, \leq U_F$	111	111	$0x1, 1x0$	000
$x_4, \leq U_F$	111	111	000	$0x1, 1x0$

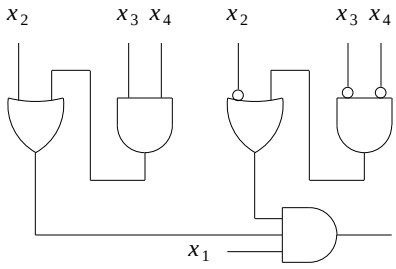


Figure 6: An example of testing a comparison unit

3.4 Identifying comparison functions

The following procedure is a straightforward method to determine whether a function f is a comparison function. For every permutation $(x_1, x_2, \cdots, x_n)$ of the variables of f, obtain the minterms where $f(x_1, x_2, \cdots, x_n) = 1$. If all these minterms have consecutive decimal numbers, then f is a comparison function. The complexity of this procedure is $O(n! \cdot 2^n)$, where n is the number of variables of f. The term $n!$ results from the worst case where all $n!$ permutations of the variables of f have to be considered. The term 2^n results from considering the minterms where $f = 1$. For functions with small numbers of inputs, this procedure accurately determines whether the function is a comparison function. It is possible to remove the $n!$ complexity term and obtain a procedure applicable to functions with larger numbers of inputs by formulating the problem as a Hamiltonian path problem and using heuristics to find an appropriate solution. Since the comparison functions we consider in our experiments are small we omit the details of this procedure.

4. Circuit optimization

In this section we consider the problem of reducing the number of gates and the problem of reducing the number of paths in a given gate-level circuit by using comparison functions.

4.1 Reducing the number of gates

The general form of the procedure is as follows. The procedure traces the circuit from the primary outputs towards the primary inputs. Every gate-output g in the circuit is considered, except for gate-outputs that become internal to comparison units already selected. For every gate-output g considered, we derive several subcircuits with output g as explained below. For each subcircuit C' with inputs I', we check whether the function $f'(I')$ that C' implements on g is a comparison function. If f' is not a comparison function, then C' is discarded. Otherwise, we compute the number of gates in a comparison unit implementing f', denoted N'. We compare this number to the number of gates currently implementing f', denoted N. In computing N, we take into account the fact that some of the gates may fan out, and thus may be common to f' and to other subfunctions. Such common gates are not included in the count N, since they cannot be removed even if C' is replaced by a comparison unit. We then select the comparison unit that results in the largest reduction in the number of gates and replace C' by it. If a choice exists, we select the comparison unit that results in the smallest number of paths on g. To ensure that a comparison function always exists and that the number of gates is never increased, the set of subcircuits considered for line g always contains a subcircuit comprised of the single gate with output g. The procedure is summarized next.

Procedure 2: Reducing the number of gates

(1) Mark all the primary outputs and unmark all other lines. Set $g = N_L$, where N_L is the number of circuit lines.

(2) If line g is a marked gate-output:

  (a) Find all candidate subcircuits with output g. For every candidate subcircuit C':
    If C' does not implement a comparison function, then eliminate C'. Otherwise, compute the reduction in the number of gates if C' is replaced by a comparison unit, $N - N'$.

  (b) For every subcircuit C' that results in the maximum reduction in the number of gates, compute the number of paths on g if C' is replaced by a comparison unit.

  (c) Select the subcircuit C' that results in the maximum reduction in the number of gates and the minimum number of paths on g.

  (d) Mark that C' is selected and mark all the inputs of C' which are not primary inputs (such lines will be considered in Step 2).

(3) Set $g = g - 1$. If $g > 0$ go to Step 2.

Next, we describe the computation of all the candidate subcircuits with a given output g. The computation starts with the subcircuit C_0 containing the gate that line g is its output. Let the i-th subcircuit obtained be C_i. Let I_i be the set of inputs of C_i. For every line $h \in I_i$ such that h is the output of gate H, we define a subcircuit $C_k = C_i \bigcup \{H\}$. If the number of inputs of C_k does not exceed a predetermined limit K, C_k is also used to generate new subcircuits. The process ends when no new subcircuits can be generated. Values of $K = 5,6$ were found to be useful in our experiments.

After Procedure 2 is applied, a new gate-level circuit is generated and Procedure 2 is applied to the new circuit. This is repeated until no additional reduction in the number of gates is possible.

4.2. Reducing the number of paths

The general form of the procedure for reducing the number of paths is similar to Procedure 2. The procedure is referred to as Procedure 3. In Procedure 3, C' is selected as the subcircuit that yields the smallest number of paths to g. We do not use the number of gates as a secondary objective in Procedure 3, since our experimental results indicate that it does not improve the results obtained. Procedure 3 is applied repeatedly until no more improvements in the number of paths are possible.

4.3 Combined measures

The circuit optimization problem considered here has two dimensions, the number of gates and the number of paths. Every circuit equivalent to the given circuit corresponds to a point in the solution space. Procedures 2 and 3 search for the extreme points in this space, corresponding to the minimum number of gates and the minimum number of paths, respectively. It is also possible to search for points in between by minimizing the number gates and the number of paths simultaneously. This can be accomplished by selecting subcircuits that yield the maximum improvement in a measure that is based both on the number of paths and on the reduction in the number of gates. Our goal in this work is to exhibit the extreme points that can be achieved and we do not pursue the possibility of optimizing both parameters at the same time.

5. Experimental results

We applied Procedures 2 and 3 to irredundant, fully-scanned ISCAS89 benchmark circuits that have more than 10,000 paths. Irredundant circuits were obtained using the procedure of [15]. Comparison functions were identified by trying up to 200 permutations of the inputs and checking whether either the minterms where the function is 1 or the minterms where the function is 0 are consecutive. In the latter case, f' was implemented as a comparison function and f was obtained by complementing the comparison unit output.

To measure the change in the number of gates due to Procedure 2, we count the number of equivalent two-input gates. A k-input gate can be implemented as an interconnection of $k-1$ 2-input gates, and therefore adds $k-1$ to the gate count. We used equivalent 2-input gates to ensure that the way in which gates with large numbers of inputs are implemented does not affect the gate count. In addition, Procedure 2 uses the number of equivalent two-input gates as its minimization criterion. For all circuits, we considered subcircuits with up to $K = 5$ and up to $K = 6$ inputs (in two separate experiments). We also considered some of the smaller circuits with $K = 7$, but in the majority of cases the results obtained were inferior to the results obtained with $K = 5$ and $K = 6$. Low values of K have the advantage that they have a small number of logic levels, thus, the total circuit delay does not increase.

Several circuits turned out to contain redundant stuck-at faults after applying Procedure 2. This is in spite of the fact that the original circuits are irredundant and that comparison units are fully testable for stuck-at faults when their inputs are independently controlled. The reason for the existence of redundant faults is as follows. Consider a subcircuit C with a set of inputs I, replaced by a comparison unit D with the same set of inputs I. Suppose that a stuck-at fault f in D requires a combination α on I. Suppose in addition that α is not required on I for any fault in C. If α cannot be obtained on I due to the logic driving it, then f is redundant in the modified circuit even if the original circuit was irredundant. Consequently, the redundancy removal procedure from [15] was applied after Procedure 2 whenever redundant faults were found.

The results of Procedure 2 followed by redundancy removal are reported in Table 2 in the following format. After circuit name, we give the value of K for which the best modified circuit was obtained. The numbers of two-input gates in the original circuit, in the best modified circuit and in the modified circuit after redundancy removal are given next. In the last column, we give the number of paths in the original circuit, in the circuit after modification and in the modified circuit after redundancy removal. The results after redundancy removal are omitted if no redundant stuck-at faults were found. It can be seen that the proposed procedure consistently reduces the number of gates as well as the number of paths. The reduction in the number of paths is often very large. Redundancy removal has a minor effect on the size of the resulting circuit, however, it is important to ensure complete stuck-at fault testability.

For comparison purposes, we give in Table 3 under the "RAMBO_C [1]" header the number of equivalent 2-input gates and the number of paths in the circuits obtained after applying the procedure of [1] to some of the circuits considered in Table 2. The procedure of [1] yields better gate reductions than the procedure proposed here. However, the number of paths in the circuits of [1] are higher in most cases, even higher than the original number of paths in three out of the four circuits used in this

Table 2: Results of Procedure 2

	2-inp. gates			paths		
circuit(K)	orig	modif	red.rem	orig	modif	red.rem
irs1423 (6)	491	490	488	42,089	37,293	37,278
irs5378 (6)	1394	1388		10,976	10,581	
irs9234 (6)	1929	1784	1783	109,283	20,333	20,330
irs13207(6)	2737	2537		261,312	85,174	
irs15850(6)	3361	3115	3107	23,003,369	3,635,532	3,584,511
irs35932(5)	9900	8497		58,645	20,898	
irs38417(5)	9698	9344	9316	1,192,971	674,081	672,121
irs38584(5)	12037	11773		565,433	157,979	

experiment. We also applied Procedure 2 after the procedure of [1] has been applied to further reduce the number of gates and at the same time reduce the number of paths. The results are given in Table 3 under the header "RAMBO_C+Proc.2". Here again Procedure 2 consistently reduces the numbers of gates and paths.

Table 3: Comparison with [1]

	orig		RAMBO_C[1]		RAMBO_C+Proc.2		
circuit	2-inp	paths	2-inp	paths	K	2-inp	paths
irs1423	491	42,089	448	54,596	6	448	50,000
irs5378	1394	10,976	1248	12,235	6	1242	11,552
irs9234	1929	109,283	1539	32,376	6	1497	23,133
irs13207	2737	261,312	2266	577,911	6	2171	163,525

The reduction in the number of gates is not a direct measure of circuit size. To obtain a better estimate of the effect of the proposed procedure on circuit size, we applied the technology mapping procedure included in SIS to the original benchmark circuits before and after Procedure 2 (cf. Table 2) and to the circuits after applying the procedure of [1] before and after applying Procedure 2 (cf. Table 3). Circuit parameters are shown in Table 4. For each set of circuits considered we show the number of literals and the number of gates on the longest path (indicating the circuit delay). It can be seen that reductions in circuit size are consistent with those of Tables 2 and 3, although the contents of the technology library was not directly considered by the proposed modification procedure. In addition, the length of the longest path through the circuit does not increase with the application of the procedure proposed here.

Table 4: Technology mapping

(a) Original circuits

	original		Proc.2	
circuit	literals	longest	literals	longest
irs1423	1035	72	1031	70
irs5378	2607	17	2610	16
irs9234	3817	30	3577	30
irs13207	5443	31	5004	31

(b) Circuits after the procedure of [1]

	RAMBO_C		RAMBO_C+Proc.2	
circuit	literals	longest	literals	longest
irs1423	959	68	956	66
irs5378	2413	20	2428	20
irs9234	3140	30	3090	30
irs13207	4591	35	4487	35

The results of applying Procedure 3 that targets reduction of the path count are shown in Table 5. It can be seen that the number of paths is reduced more than in Table 2, however, sometimes at the cost of increasing the number of gates.

Next, we report the effect of the proposed modifications on the testability of the resulting circuits. Due to space considerations, we consider only circuits synthesized by Procedure 2

Table 5: Results of Procedure 3

circuit(K)	inp	out	2-inp. gates orig	2-inp. gates modif	paths orig	paths modif
irs1423 (6)	91	79	491	503	42,089	35,810
irs5378 (6)	214	224	1394	1476	10,976	9,746
irs9234 (6)	247	248	1929	1981	109,283	19,842
irs13207(6)	699	788	2737	2606	261,312	85,151
irs15850(6)	611	680	3361	3690	23,003,369	2,875,815
irs35932(5)	1763	2048	9900	10850	58,645	20,898
irs38417(6)	1664	1742	9698	10825	1,192,971	624,779
irs38584(6)	1455	1700	12139	11953	565,433	156,201

followed by redundancy removal using the procedure from [15]. We report only their random pattern testability for stuck-at faults and for path delay faults.

To determine the stuck-at fault testability, we applied up to 30,000,000 random patterns to the circuits under consideration. We used the fault simulator FSIM [17] to obtain the fault coverage. The results are reported in Table 6, as follows. After circuit name, we give the number of stuck-at faults, the number of faults that remained undetected after applying 30,000,000 random patterns, and the last pattern that was effective in detecting any fault in the original circuit. Then, we give the same information for the modified circuit after redundancy removal. It can be seen that the random pattern testability for stuck-at faults remained unchanged after the modifications.

Table 6: Results for stuck-at faults

circuit	original faults	original remain	original eff.patt	modified faults	modified remain	modified eff.patt
irs1423	1468	0	34656	1439	0	34656
irs5378	4500	0	114848	3515	0	114848
irs9234	5768	0	15606336	4672	0	15606336
irs13207	8813	0	333120	7452	0	333120
irs15850	10510	18	27884608	8795	16	27884608
irs35932	33174	0	256	26595	0	256
irs38417	30472	0	9485440	26002	0	9485440
irs38584	33536	9	25454368	30802	9	25454368

Next we considered the robust path delay fault testability of the circuits. The results of applying random patterns until no change in fault coverage was obtained for 100,000 consecutive random patterns are reported in Table 7 for the four versions of *irs* 13207 considered in Tables 2 and 3. The last effective pattern (i.e., the last pattern that detected any fault) is given under column "eff". It occurred after the same number of patterns for each pair of circuits, before and after modification. The number of faults detected and the total number of faults are given in this order under column "det/faults". We conclude that when the number of path delay faults was reduced by Δ, the number of undetected path delay faults was reduced by more than Δ. Thus, the fault coverage increased significantly.

Table 7: Robust detection by random patterns in *irs* 13207

circuit	eff	det/faults original	det/faults modified
original	131,000	7,304/522,624	8,324/170,348
RAMBO_C	132,000	7,459/1,155,822	8,096/327,050

6. Concluding remarks

We defined a class of functions called comparison functions and showed how they can be implemented using comparison units. Comparison units are fully testable for stuck-at faults and for path delay faults. In addition, comparison units have a small number of paths through them. We proposed a method of modifying a given circuit to reduce its size and enhance its path delay fault testability by replacing subcircuits of the given circuit with comparison units. Experimental results showed moderate reductions in the number of gates and significant reductions in the number of paths in the circuits considered. It was shown that most of the path delay faults removed were untestable by random patterns. The random pattern testability properties of the circuits to stuck-at faults remained unchanged.

Several issues remain to be investigated. (1) Combinations of values that cannot be obtained due to logic dependencies in the circuit can be used during the selection of comparison units to ensure that all the faults in each comparison unit can be tested. (2) Synthesis using multiple comparison units to replace a given subcircuit can help reduce the number of gates and/or paths even further.

References

[1] K.-T. Cheng and L. A. Entrena, "Multi-Level Logic Optimization by Redundancy Addition and Removal", 1993 Europ. Conf. on Design Autom., Feb. 1993.

[2] W. Kunz and P. Menon, "Multi-Level Logic Optimization by Implication Analysis", in Proc. 1994 Intl. Conf. on Computer-Aided Design.

[3] S. Muroga, Y. Kambayashi, H. C. Lai and J. N. Culliney, "The Transduction Method - Design of Logic Networks Based on Permissible Functions", IEEE Trans. on Computers, Oct. 1989, pp. 1404-1424.

[4] K. Roy, K. De, J. A. Abraham, and S. Lusky, "Synthesis of delay fault testable combinational logic," in Proc. Int. Conf. on Computer-Aided Design, Nov. 1989, pp. 418-421.

[5] I. Pomeranz and S. M. Reddy, "On the Number of Tests to Detect All Path Delay Faults in Combinational Logic Circuits", Technical Report No. 12-1-1992.

[6] H. Hengster, R. Drechsler and B. Becker, "Testability Properties of Local Circuit Transformations with respect to the Robust Path-Delay-Fault Model", in Proc. 7th Int. Conf. on VLSI Design, Jan. 1994, pp. 123-126.

[7] J. D. Lesser and J. J. Schedletsky, "An experimental delay test generator for LSI logic," IEEE Trans. Comput., vol. C-29, pp. 235-248, Mar. 1980.

[8] I. Pomeranz and S. M. Reddy, "An Efficient Non-Enumerative Method to Estimate Path Delay Fault Coverage", Proc. Intl. Conf. on Computer-Aided Design, 1992, pp. 560-567.

[9] C. J. Lin and S. M. Reddy, "On delay fault testing in logic circuits," IEEE Trans. CAD, pp. 694-703, Sept. 1987.

[10] I. Pomeranz, S. M. Reddy and P. Uppaluri, "NEST: A Non-Enumerative Test Generation Method for Path Delay Faults in Combinational Circuits", in Proc. 30th Design Autom. Conf, 1993, pp. 439-445.

[11] W. K. Lam, A. Saldanha, R. K. Brayton, A. L. Sangiovanni-Vincentelli, "Delay Fault Coverage and Performance Tradeoffs", in Proc. 30th Design Autom. Conf., 1993, pp. 446-451.

[12] K.-T. Cheng and H.-C. Chen, "Delay Testing for Non-Robust Untestable Circuits", in Proc. Intl. Test Conf., Oct. 1993, pp. 954-961.

[13] I. Pomeranz and S. M. Reddy, "Design-for-Testability for Path Delay Faults in Large Combinational Circuits Using Test-Points", 31st Design Autom. Conf., June 1994, pp. 358-364.

[14] E. J. McCluskey, *Logic Design Principles with Emphasis on Testable Semicustom Circuits*, Prentice-Hall, 1986.

[15] S. Kajihara, H. Shiba and K. Kinoshita, "Removal of Redundancy in Logic Circuits under Classification of undetectable Faults", Proc. 22nd Fault-Tolerant Computing Symp., July 1992, pp. 263-270.

[16] C.-H. Chiang and S. K. Gupta, "Random Pattern Testable Logic Synthesis", in Proc. 1994 Intl. Conf. on Computer-Aided Design, Nov. 1994, pp. 125-128.

[17] H. K. Lee and D. S. Ha, "An Efficient, Forward Fault Simulation Algorithm based on the Parallel Pattern Single Fault Propagation", in Proc. 1991 Intl. Test Conf., Oct. 1991, pp. 946-955.