Logic Synthesis Techniques for Reduced Area Implementation of Multilevel Circuits with Concurrent Error Detection

Nur A. Touba and Edward J. McCluskey

Center for Reliable Computing
Departments of Electrical Engineering and Computer Science
Stanford University, Stanford, CA 94305-4055

Abstract

This paper presents new logic synthesis techniques for generating multilevel circuits with concurrent error detection based on a parity-check code scheme that can detect all errors caused by single stuck-at faults. These synthesis techniques fully automate the design process and allow for a better quality result than previous methods thereby reducing the cost of concurrent error detection. An algorithm is described for selecting a good parity-check code for encoding the outputs of a circuit. Once the code has been chosen, a new procedure called structure-constrained logic optimization is used to minimize the area of the circuit as much as possible while still using a circuit structure that ensures that single stuck-at faults cannot produce undetected errors. The implementation that is generated is path fault secure and when augmented by a checker forms a self-checking circuit. Results indicate that self-checking multilevel circuits can be generated which require significantly less area than using duplication.

1. Introduction

Concurrent error detection is an important requirement in the design of systems in which reliability and data integrity are important. Concurrent error detection circuitry has the ability to detect both transient and permanent faults as well as to enhance off-line testability and reduce BIST overhead [5], [14].

One general approach for concurrent error detection is to encode the outputs of a circuit with an error detecting code and have a checker that monitors the outputs and gives an error indication if a non-codeword occurs. A *systematic code* is a code in which codewords are constructed by augmenting the normal output bits with check bits. Using a systematic code for concurrent error detection has the advantage that no decoding is needed to get the normal output bits. Figure 1 shows the general structure of a circuit being checked with a systematic code. There are three parts: function logic, check symbol generator, and checker. The function logic generates the normal outputs, the check symbol generator generates the check bits, and the checker determines if they form a codeword. Two types of systematic codes that are used for concurrent error detection are Berger codes and parity-check codes.

The conventional approach for designing arbitrary multilevel circuits with concurrent error detection has been

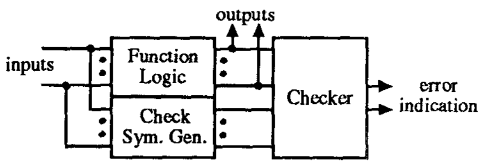

Fig. 1. Concurrent Error Detection using a Systematic Code

to use duplication. The circuit is simply duplicated and the outputs are compared using an equality checker. While this provides very high error detection capability, it has a large area overhead. Recently, research has been done on using automated logic synthesis techniques (such as those used in MIS [1]) to design multilevel circuits with concurrent error detection requiring less area overhead than duplication while still detecting all errors due to internal single stuck-at faults [3], [7]. *Internal single stuck-at faults* are all single stuck-at faults except those at the primary inputs (PI's). Note that for any concurrent error detection scheme (including duplication), detection of stuck-at faults at the PI's cannot be guaranteed unless encoded inputs are used. However, if the inputs to the circuit are outputs of another concurrently checked logic block, then the only undetectable PI faults are break faults after the checker [10].

Jha and Wang [7] described a method for synthesizing self-checking circuits based on a unidirectional code (e.g., a Berger code). De *et al.* [3] described the RSYN synthesis system which can synthesize self-checking circuits based on one of three schemes: Berger code, parity-check code, or duplication. They gave results for some benchmark circuits, and for almost all of the circuits, the parity-check code scheme required the least area overhead. This paper presents new logic synthesis techniques for generating multilevel circuits with concurrent error detection based on parity-check codes. These techniques allow for a significant improvement in the quality of the result.

A *parity-check code* is a code in which each check bit is a parity check for a group of output bits. Each group of outputs that is checked by a parity check bit is called a *parity group.* In single-bit parity, there is one parity group which contains all of the outputs. In duplication of a circuit with n outputs, there are n parity groups each containing one of the outputs. The two basic steps in synthesizing a circuit that uses a parity-check code for concurrent error detection are: (1) determining which parity-check code to use, and (2) performing logic optimization under the constraint that the structure of the

©1994 ACM 0-89791-690-5/94/0011/0651 $3.50

circuit is such that faults cause detectable errors (i.e., non-codeword outputs). In this paper, a fully automated parity-check code selection algorithm is presented. It is a greedy algorithm that tries to find the optimal code for minimizing the overall area of the circuit using a cost function that considers the area of all three parts of the circuit: function logic, parity predict logic (check symbol generator), and checker. Once the code is selected, the next step is to perform logic optimization under structural constraints. In this paper, a new logic optimization technique called *structure-constrained logic optimization (SCLO)* is presented. By considering structural constraints when factoring, SCLO optimizes the area of the circuit as much as possible under the constraints.

2. Terminology

A multilevel circuit can be represented by a *Boolean network* which is a directed acyclic graph where each node corresponds to a Boolean function and each inward arc indicates an input to the function. If a directed path exists from node i to node j, then node i is a *transitive fan-in* of node j, and node j is a *transitive fan-out* of node i.

The *totally self-checking (TSC) goal* is to detect the first error that occurs due to any fault in a specified fault class. The concept of path fault secure (PFS) circuits was introduced in [15] and is defined as follows.

Definition 1: A circuit is *path fault secure (PFS)* if and only if for every fault in a specified fault class, error propagation down any set of possible structural paths from the fault site to the outputs will never produce an incorrect codeword output.

It was shown in [15] that PFS circuits that are checked by a TSC checker achieve the TSC goal regardless of which input patterns occur during normal operation. The techniques described in this paper generate self-checking circuits that are PFS.

3. Selecting Parity-Check Code

Given a combinational circuit for which concurrent error detection is required, the first step is to select a parity-check code for encoding the outputs. To make the circuit PFS for all internal single stuck-at faults, logic cannot be shared between two outputs in the same parity group because then if a fault occurred in the shared logic, an error could propagate to both outputs causing a two bit error which would not be detected by the parity checker. Therefore, a tradeoff exists between the number of parity groups (i.e., check bits), and the constraints on logic sharing between outputs. The more parity groups there are, the more logic sharing is possible, however more parity groups require more parity predict logic to generate the check bits.

The goal is to select the code that will require the least area to implement. The area of the circuit is equal to the sum of the areas of the function logic, parity predict logic, and checker. The area required by the function logic depends on how much logic sharing is possible. The area required by the parity predict logic depends on the size of the parity functions that must be implemented for each check bit. The area required by the checker depends on how many parity groups there are.

Table 1. Algorithm for Selecting Parity-Check Code

```
SELECT_PARITY_CHECK_CODE (funct_logic):
  opt_funct_logic = unconstrained logic opt. of funct_logic
  best_code = duplication code
  /* one parity group for each output */
  for $i$ = 1 to NUM_PO(funct_logic)
      $parity\_group_i = \{ i \}$
  repeat {
      for each $code_{i,j}$ formed by combining parity
        groups $i$ and $j$ in best_code
          compute AREA_REDUCE(best_code, $code_{i,j}$)
      $code_{p,q}$ = code that maximizes AREA_REDUCE
      if ( AREA_REDUCE(best_code, $code_{p,q}$) > 0 ) {
          best_code = $code_{p,q}$
          /* combine parity groups $p$ and $q$ */
          $parity\_group_p = parity\_group_p \cup parity\_group_q$
          delete $parity\_group_q$
          improve = true
      } else improve = false
  } until ( ! improve || best_code == single-bit parity code )
  return ( best_code )
AREA_REDUCE (best_code, $code_{i,j}$):
  $shared_{i,j}$ = LITS_SHARED(opt_funct_logic,
                        $parity\_group_i$, $parity\_group_j$)
  $c_{i,j}$ = SIMPLIFY( XOR($c_i$, $c_j$) )
  $parity\_reduce_{i,j}$ = LITS($c_i$) + LITS($c_j$) - LITS($c_{i,j}$)
  $checker\_reduce_{i,j}$ = 4
  return ( $parity\_reduce_{i,j} + checker\_reduce_{i,j} - shared_{i,j}$ )
```

The number of possible parity-check codes for a circuit with n outputs is equal to B_n, the number of partitions of a set of n objects. This number is exponential in n, so heuristics are needed in searching for the minimal area code. A greedy algorithm is given in Table 1 which uses the heuristic of pairwise combining parity groups in searching for a minimal area code. It begins with the duplication code in which each output is in its own parity group. It then estimates the reduction in the area for all codes that can be formed by combining two of the parity groups and chooses the code that offers the largest area reduction. This process continues until no further reduction in area is possible through combining parity groups.

The area reduction for using code B instead of code A, where code B is formed by combining two parity groups in code A, is estimated by considering the resulting change in the area for each of the three parts of the circuit:

Function Logic: Area may increase because of added constraints on logic sharing between outputs. This is estimated by looking at the function logic optimized with no constraints and computing the literal count of the logic shared between the outputs that are combined into the same parity group in code B since this logic can no longer be shared if code B is used instead of code A. However, global restructuring operations during SCLO may compensate for some of this loss by factoring the circuit differently.

Parity Predict Logic: Area will decrease because there will be one less parity function. Let c_1 and c_2 be the parity functions for the parity groups in code A that are combined to form code B. The number of literals that will be saved is estimated by comparing the factored form literal count of c_1 plus c_2 with that of $(c_1 \oplus c_2)$ after simplifying:

$$\text{p.p.l. area}(A) - \text{p.p.l. area}(B) \approx \text{lits}(c_1) + \text{lits}(c_2) - \text{lits}(c_1 \oplus c_2)$$

Checker: Area will decrease because there is one fewer check bit. The checker will have 4 fewer literals.

In the worst case, the code selection algorithm will execute the AREA_REDUCE function $\sum_{i=1}^{n} C_2^i$ times, where n is the number of PO's. Thus, a solution is obtained using only $O(n^2 \log_2 n)$ operations, where each operation requires computing the exclusive OR of two functions and simplifying it.

4. Structure-Constrained Logic Optimization (SCLO)

Once the parity-check code has been selected, the next step is to optimize the circuit under the constraints on logic sharing that are needed to ensure that no internal single stuck-at fault can cause an undetectable error. A new synthesis technique that does this, called *structure-constrained logic optimization (SCLO)*, is described here. Multilevel logic optimization improves circuit area by using operations that restructure and minimize the logic represented by a Boolean network. In SCLO, restrictions are placed on the restructuring operations to ensure that the resulting circuit will satisfy the structural constraints.

Given the initial multilevel logic equations and the parity-check code, SCLO optimizes the logic under the constraint that a non-PI node cannot be a transitive fan-in of more than one PO in a parity group. SCLO is accomplished by starting with an initial Boolean network that satisfies the constraints, and then constraining the restructuring operations so that they never cause nodes to violate the constraints. The two restructuring operations that can cause a node to violate the constraints are resubstitution and extraction.

4.1 Constraints on Resubstitution

Resubstitution is an operation where some node a, which is "divisible" by another node b, is rewritten as a function of node b, thus creating an arc from node b to node a [1]. This arc may create a path such that node b (or some node that is a transitive fan-in of node b if Boolean resubstitution is considered) becomes a transitive fan-in of more than one PO in a parity group, thus violating the constraints. Therefore, in SCLO, resubstitution can be performed between two nodes only if the resulting arc does not violate the constraints.

Various filters are generally used to reduce the number of node pairs for which resubstitution is attempted [1]. So this constraint can be simply added as an additional filter.

4.2 Constraints on Extraction

Extraction is an operation in which an intermediate node is created by factoring out a common subexpression from a set of nodes, S. The intermediate node will have an arc to each node in S, and hence will be a transitive fan-in of every PO that is a transitive fan-out of any node in S. Therefore, in SCLO, common subexpressions can only be extracted from a set of nodes if all the PO's that are transitive fan-out's of the set of nodes are in different parity groups.

In order to implement constrained extraction, the process of selecting common subexpressions to extract needs to be modified. Two methods that are used for selecting common subexpressions to extract are rectangle covering [2] and the concurrent decomposition procedure in [12]. In both methods, subexpressions are identified, and each is assigned a value based on the number of literals that will be reduced if it is extracted. In SCLO, subexpressions cannot always be extracted from the full set of possible nodes due to the structure constraints. Thus, the value assigned to each subexpression must be adjusted according to the maximum number of literals that can be reduced without violating the constraints. Subexpressions can then be selected based on the adjusted values and extracted from the maximal set of nodes that the constraints allow. Details on a procedure for extraction in SCLO using rectangle covering can be found in [16].

4.3 Technology Mapping

After the Boolean network is optimized, a technology mapping procedure that follows the structure of the Boolean network, such as tree-mapping [4], [8], is used to map the Boolean network to single-output library cells. The resulting mapped circuit is PFS for internal single stuck-at faults because each such fault affects only one cell and can propagate to no more than one output in any parity group using any set of possible structural paths. Thus, it can never produce an incorrect codeword output.

For a self-checking circuit, a TSC checker needs to be added to the PFS circuit generated by SCLO. TSC parity checkers [9] can be used to check each parity group, and then a TSC two-rail checker [6] can be used to combine the error indication signals.

5. Results

The synthesis method proposed in this paper has been implemented by making modifications to SIS 1.1 (an updated version of MIS). The code selection algorithm was added, and the restructuring algorithms were extended to handle structural constraints so that SCLO could be performed. Self-checking circuits were generated for some of the MCNC combinational benchmark circuits and then placed and routed using the TimberwolfSC 4.2c standard cell package [13]. The circuits were optimized using the script file **script.boolean** included with the SIS source code. Results are shown in Table 2 for some of the circuits which were chosen to illustrate the variance in the number of check bits for the parity-check code selected by the code selection algorithm. Literal counts are given in

Table 2. Results for MCNC Benchmark Circuits

Circuit					Duplication Method						Synthesis Method					
Name	PI	PO	opt lits	layout area	chk bits	ckt lits	chkr lits	total lits	layout area	ovrhd %	chk bits	ckt lits	chkr lits	total lits	layout area	ovrhd %
5xpl	7	10	121	334	10	242	72	314	989	196	3	217	44	261	797	139
alu4	14	8	800	3298	8	1600	56	1656	6796	106	8	1600	56	1656	6796	106
b12	15	9	87	284	9	174	64	238	727	156	4	150	44	194	613	116
bw	5	28	217	752	28	434	216	650	2566	241	8	302	136	438	1066	42
cmb	16	4	52	153	8	104	24	128	414	171	2	64	16	80	206	35
cu	14	11	53	181	11	106	80	186	537	197	1	83	40	123	356	97
f51ml	8	8	130	385	8	260	56	314	923	140	2	218	32	250	677	76
misex1	8	7	54	154	7	108	48	156	403	162	3	96	32	128	355	131
misex2	25	18	104	372	18	208	136	344	1166	213	2	202	72	278	935	151
pcle	19	9	69	229	9	138	64	202	659	188	3	156	40	196	523	128
sao2	10	4	149	431	4	298	24	322	1076	150	1	268	12	280	785	82
term1	34	10	179	617	10	358	72	430	1542	150	7	326	60	386	1211	96
tt2	24	21	191	630	21	382	160	542	1977	220	9	374	112	486	1707	171
x2	10	7	51	144	7	102	48	150	433	201	2	79	28	107	279	94

terms of factored form literals and layout areas are given in units of 1000 λ^2, where λ is the minimum size in a technology. Under the first major heading, information about each circuit is given: number of PI's, number of PO's, literal count after normal unconstrained logic optimization, and layout area for the optimized circuit (with no concurrent error detection). Under the second and third major headings, results for the duplication method and the synthesis method proposed in this paper are given: number of check bits, literal count for the circuit, literal count for the checker, total literal count for the self-checking circuit, layout area for the self-checking circuit, and the percentage of area overhead required which is computed as show below.

$$\%\text{ Area Overhead} = \frac{\text{(self-checking layout area) - (normal layout area)}}{\text{(normal layout area)}} \times 100$$

For the circuits where the number of check bits (i.e., parity groups) that are used in the parity-check code chosen by the synthesis method is equal to the number of PO's, the duplication code was selected. Where the number of check bits is equal to one, single-bit parity prediction was selected.

6. Conclusions

The logic synthesis techniques presented here produce a better result than previous synthesis methods because the parity-code selection algorithm uses a cost function based on the area of the function logic, parity predict logic, and checker, and the structure-constrained logic optimization technique considers the structural constraints during each step of logic optimization. Results were presented that show that these techniques can significantly reduce the area overhead required for concurrent error detection in multilevel circuits. A possibility for future research is to apply these techniques to synthesis of fault-tolerant finite state machines where the states are encoded with a parity-check code [11]. Also, the structure-constrained logic optimization procedure described here can easily be generalized for any types of structural constraints during logic synthesis and may have other applications.

References

[1] Brayton, R.K., R. Rudell, A. Sangiovanni-Vincentelli, A.R. Wang, "MIS: A Multiple-Level Logic Optimization System," *IEEE Trans. Comp.-Aided Design*, pp. 1062-1081, Nov. 1987.

[2] Brayton, R.K., R. Rudell, A. Sangiovanni-Vincentelli, A.R. Wang, "Multi-Level Logic Optimization and The Rectangular Covering Problem," *Proc. of ICCAD*, pp. 66-69, 1987.

[3] De, K., C. Natarajan, D. Nair, and P. Banerjee, "RSYN: A System for Automated Synthesis of Reliable Multilevel Circuits," *IEEE Transactions on VLSI Systems*, pp. 186-195, Jun. 1994.

[4] Detjens, E., G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Technology Mapping in MIS," *Proc. of Int. Conf. on Computer-Aided Design (ICCAD)*, pp. 116-119, 1987.

[5] Gupta, S.K., and D.K. Pradhan, "Can Concurrent Checkers Help BIST?," *Proc. of International Test Conf.*, pp. 140-150, 1992.

[6] Hughes, J.L.A., E.J. McCluskey, and D.J. Lu, "Design of Totally Self-Checking Comparitors with an Arbitrary Number of Inputs," *IEEE Transactions on Computers*, pp. 546-550, Jun. 1984.

[7] Jha, N.K., and S.-J. Wang, "Design and Synthesis of Self-Checking VLSI Circuits and Systems," *IEEE Transactions on Computer-Aided Design*, pp. 878-887, Jun. 1993.

[8] Keutzer, K., "Dagon: Technology Binding and Local Optimization by DAG Matching," *Proc. of the 24th Design Automation Conference*, pp. 341-347, 1987.

[9] Khakbaz, J., "Self-Testing Embedded Parity Trees", *Proc. of FTCS-12*, pp. 109-116, 1982.

[10] Khodadad-Mostashiry B., "Break Faults in Circuits with Parity Prediction," *Tech. Note No. 183*, CRC, Stanford University, Stanford, CA, Dec. 1980.

[11] Leveugle R., "Optimized State Assignment of Single Fault Tolerant FSMs Based on SEC Codes", *Proc. of the 30th Design Automation Conference*, pp.-14-18, 1993.

[12] Rajski, J., and J. Vasudevamurthy, "The Testability-Preserving Concurrent Decomposition and Factorization of Boolean Expressions," *IEEE Trans. on CAD*, pp. 778-793, Jun. 1992.

[13] Sechen C., and A. Sangiovanni-Vincentelli, "TimberWolf3.2: A New Standard Cell Placement and Global Routing Package", *Proc. of the 30th Design Automation Conference*, pp. 432-439, 1986.

[14] Sedmak, R.M., "Design for Self-Verification: An Approach for Dealing with Testability Problems in VLSI-Based Designs", *Proc. of International Test Conference*, pp. 112-120, 1979.

[15] Smith, J.E., and G. Metze, "Strongly Fault Secure Logic Networks," *IEEE Trans. on Computers*, pp. 491-499, Jun. 1978.

[16] Touba, N.A., and E.J. McCluskey, "Logic Synthesis for Concurrent Error Detection", *Technical Report No. 93-6*, CRC, Stanford University, Stanford, CA, Nov. 1993.