A Synthesis Procedure for Flexible Logic Functions

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

1. Introduction

In most applications of digital logic circuits, the circuit function is either specified (0,1) or unspecified (*don't – care*) for every input condition. However, there are also applications where any one of a subset of functions is an acceptable solution, even though it is not possible to represent all the functions in terms of output don't-cares. In this case, we say that the function is *flexible*. Flexible functions were considered before in [1]. In this work, we propose a synthesis procedure for flexible functions based on functional blocks called comparison units [2]. The main differences between the proposed procedure and the procedures of [1] are the following. (1) We do not require a closed-form representation of all the flexibility that exists in specifying the function f. We only require that a procedure would exist to check whether a given function belongs to the class of acceptable functions. (2) We use a specific architecture for the implementation of flexible functions. This architecture, based on comparison units [2], is particularly suitable for implementing flexible functions, since the correspondence between circuit size and certain properties of the implemented function is strong and easy to utilize for the minimization of the implementation. The proposed synthesis procedure starts from an acceptable function f' that may be used to implement f. It then modifies f' so as to change certain properties of f' that lead to smaller comparison unit based implementations. Before any modification of f' is accepted, a check is made to make sure that the modified function is an acceptable implementation of f. Modifications are made as long as it is possible to change the properties of f' that lead to a reduction in the implementation size.

We also demonstrate that implementations using comparison units for conventional, non-flexible functions are an effective intermediate step for synthesis. For this purpose, we apply the synthesis tool suite SIS from the University of California at Berkeley in two ways. (1) To a comparison unit implementation of a function, and (2) directly starting from the truth table of the function. In most cases, the area of the circuit derived from a comparison unit based implementation is smaller.

2. Application problem and implementation

The application we consider is that of synthesizing an autonomous finite-state machine as a test pattern generator (*TPG*) for a given circuit [3]. In the configuration we consider, the *TPG* drives a shift-register, and the inputs of the circuit-under-test (*CUT*) are driven from the shift-register. The output sequence S generated by the *TPG* is designed to produce, when shifted into the shift-register and applied to the *CUT*, a test set that detects every detectable fault in the *CUT*.

To implement a *TPG* with an output sequence S of length k, we use a counter with $N_{FF} = \lceil \log_2 k \rceil$ flip-flops. When the counter is in the state with state assignment i, the *TPG* output value $z(i)$ is equal to the value s_i in position i of S for $0 \leq i < k$. In addition, $z(i) = -$ for $k \leq i < 2^{N_{FF}}$. This determines the truth table of z as a function of the present state variables of the counter, $y_1, y_2, \cdots, y_{N_{FF}}$. Synthesis of the truth table is described next.

The basic building blocks for implementing a function in [2] are referred to as *comparison blocks*. There are two types of comparison blocks, the $\geq L$ block and the $\leq U$ block. A $\geq L$ block produces the output 1 when supplied with an input combination whose decimal value is larger than or equal to L. A $\leq U$ block produces the output 1 when supplied with an input combination whose decimal value is smaller than or equal to U. A comparison function is a function whose minterms $\{m\}$ satisfy $L \leq m \leq U$ for some L and U [2]. A comparison function can be implemented by ANDing a $\geq L$ block and a $\leq U$ block. This structure is called a *comparison unit* [2].

The minterms of a function may not all fall within a consecutive range $[L, U]$. For a function with r ranges of consecutive 1s, $\{[L_i, U_i]: 1 \leq i \leq r\}$, we OR the outputs of r comparison units where the ith one implements the range $[L_i, U_i]$.

Given the comparison unit based implementation as the target of the synthesis procedure for *TPG*s, the number of gates required to implement the *TPG* can be reduced if one can perform one of two modifications to the output function z, and consequently to the output sequence S. (1) Replace two ranges of consecutive 1s, $[L_i, U_i]$ and $[L_{i+1}, U_{i+1}]$, by a single range $[L_i, U_{i+1}]$. This implies complementing z in the range $[U_i + 1, L_{i+1} - 1]$. (2) Remove a range $[L_i, U_i]$ of consecutive 1s by complementing z from L_i to U_i.

To determine whether or not a modification can be performed in the context of the *TPG* problem, we must determine its effects on the fault coverage of the *TPG* output sequence. If the modification does not reduce the fault coverage, it is accepted, resulting in a reduction in the number of comparison blocks needed to implement the function.

3. The synthesis procedure

The definition of a comparison function in [2] allows the inputs of the function to be permuted such that the minterms create a consecutive sequence of 1s (or 0s in the case of a complemented comparison function). In this section, we use input permutations to reduce the number of ranges of consecutive 1s (0s) in z, thus reducing the size of the logic realizing the *TPG* in the application considered here.

After permuting the inputs, the don't-cares that result from the fact that the original length of S was smaller than $2^{N_{FF}}$ are interleaved with the other values. Therefore, the synthesis procedure must first determine their values. We use the following notation to describe the determination of the don't-cares.

We represent the permuted input combinations by their decimal values. For combination i, the *TPG* output value is denoted by $z(i)$. The number of counter flip-flops is denoted by N_{FF}, making the last input combination $2^{N_{FF}} - 1$. Suppose that the output values for the permuted input combinations in the range $[i_1, i_2]$ are don't-cares. The don't-care values are determined using one of the following rules. (1) If $i_1 = 0$, we set $z(i) = z(i_2 + 1)$ for $0 \leq i \leq i_2$. This causes the range of don't-

+ Research supported in part by NSF Grant No. MIP-9357581, and by NSF Grant No. CDA-9601503.

cares to be merged with the range that follows it. We obtain a new range that has the all-0 pattern as its lower bound. Thus, its $\geq L$ comparison block does not require any gates. (2) If $i_2 = 2^{N_{FF}} - 1$, we set $z(i) = z(i_1 - 1)$ for $i_1 \leq i \leq 2^{N_{FF}} - 1$. This causes the range of don't-cares to be merged with the range that precedes it. We obtain a new range that has the all-1 pattern as its upper bound. Thus, its $\leq U$ comparison block does not require any gates. (3) If $z(i_1 - 1) = z(i_2 + 1)$, we combine the range that ends at $i_1 - 1$ with the range that starts at $i_2 + 1$ by setting $z(i) = z(i_1 - 1)$ for $i_1 \leq i \leq i_2$. (4) If $z(i_1 - 1) \neq z(i_2 + 1)$, we select an index i_0, and then set $z(i) = z(i_1 - 1)$ for $i_1 \leq i \leq i_0$ and $z(i) = z(i_2 + 1)$ for $i_0 + 1 \leq i \leq i_2$. We select i_0 such that the comparison blocks $\leq i_0$ and $\geq i_0 + 1$ would require a minimum number of gates. This is achieved by selecting i_0 that ends with the maximum number of 1s [2].

The truth-table obtained can be further modified to eliminate some of the ranges. This requires us to derive the new *TPG* output sequence S' from z by first undoing the input permutation. By simulating S', we can find out whether the patterns it applies to the *CUT* detect all the faults in the *CUT*, and the modification can be accepted.

To select a permutation of the inputs, we try the original order of the counter flip-flops and 99 randomly selected permutations. For each one, we compute the initial number of ranges after setting the don't-care values. We select the original permutation, and 10 permutations that have the smallest number of ranges (possibly including the original permutation, for a total of 10 or 11 permutations). Each one of the selected permutations goes through synthesis. The permutation that yields the smallest number of gates is then selected.

4. Experimental results

We applied the procedure described above to ISCAS-85 benchmark circuits. The test sets for the circuits are the ones generated by the procedure of [4]. After applying the proposed procedure, we applied MIS-II followed by the technology mapping procedure in SIS. This reduces the size of the *TPG* by sharing gates among its various comparison units. In Table 1, after circuit name we show the initial length of S and the number of counter flip-flops. Using the best permutation of the inputs, we show the number of ranges and the corresponding number of gates obtained after applying SIS to the comparison unit based implementation. These are shown under column *proposed*, subcolumns *rang* and *SIS*. All the gate counts are given in terms of two-input gates. Under column *proposed* subcolumn *w. count* we show the number of *TPG* gates when the gates to implement the counter are taken into account.

Table 1: Experimental results

			proposed				SIS	SIS
circuit	S-len	FF	rang	gates	SIS	w.count	t.tab	FSM
c432	440	9	5	69	53	70	71	397
c499	461	9	5	60	50	67	72	376
c880	597	10	11	148	102	121	130	NA
c1355	2034	11	7	124	106	127	142	NA
c1908	3163	12	37	605	380	403	NA	NA
c3540	4248	13	151	2228	839	863	NA	NA
c6288	103	7	2	19	14	27	16	120

For comparison, we applied ESPRESSO followed by MIS-II and the technology mapping procedure in SIS to the truth tables obtained before going through the comparison unit based implementation. The gate counts obtained are shown in Table 1 under column *SIS t.tab* (these numbers do not include the counter gates). They should be compared to the results obtained by the proposed method, reported under column *proposed* subcolumn *SIS* of Table 1. From the comparison we conclude that the proposed implementation using comparison units provides a better starting point for SIS than the truth tables.

To further assess the effectiveness of the proposed synthesis procedure, we used SIS to synthesize finite-state machines that produce the initial sequences S. The number of two-input gates is shown in Table 1 under column *SIS FSM*. The number of gates should be compared to the number of gates obtained by the proposed procedure under column *proposed* subcolumn *w. count*. The proposed synthesis procedure yields significantly smaller *TPG*s than the conventional synthesis approach. In addition, the conventional approach cannot always be applied if the state table obtained for the *TPG* is too large.

From the comparison with SIS, one may view the proposed synthesis procedure and the proposed architecture based on comparison units as effective guidelines in selecting an implementation for a logic function. To further demonstrate the advantages of comparison unit based implementations as part of the synthesis flow, we considered conventional (non-flexible) functions. For each function, we followed two synthesis paths. (1) We obtained a comparison unit based implementation by permuting the inputs as above, and selecting one of the best permutations to define the ranges for the comparison units. We then applied SIS to minimize the implementation. (2) We applied ESPRESSO followed by SIS to the truth table (without going through a comparison unit based implementation).

We considered 15 random functions each having 10 inputs and one output. The probability of a 1 on the output is 0.5 for the first five functions, 0.25 for the next five functions, and 0.125 for the last five functions. The results are reported in Table 2 as follows. Each one of the 15 random functions is represented by a number. For each function, we show the number of ranges, and the number of gates after SIS (Synthesis option 1 above). In the last column we show the results of applying ESPRESSO and SIS to the truth tables directly (Synthesis option 2 above). In most cases, using comparison unit based implementations is preferable to direct synthesis.

Table 2: Results for non-flexible functions

	proposed		SIS		proposed		SIS
f	rang	SIS	t.tab	f	rang	SIS	t.tab
1	240	573	663	6	183	564	520
2	237	598	647	7	178	505	533
3	243	564	661	8	178	503	523
4	237	591	641	9	186	474	528
5	233	578	679	10	190	491	550
		2904	3291			2537	2654

	proposed		SIS
f	rang	SIS	t.tab
11	116	299	393
12	98	401	333
13	111	365	388
14	102	279	361
15	108	285	388
		1629	1863

References

[1] E. Sentovich, V. Singhal and R. Brayton, "Multiple Boolean Relations", Intl. Workshop on Logic Synthesis, 1993.

[2] I. Pomeranz and S. M. Reddy, "On Synthesis-for-Testability of Combinational Logic Circuits", 32nd Design Automation Conf., June 1995, pp. 126-132.

[3] V. D. Agrawal, C. R. Kime and K. K. Saluja, "A Tutorial on Built-In Self-Test Part 1: Principles", IEEE Design and Test of Computers, March 1993, pp. 73-82.

[4] S. Kajihara, I. Pomeranz, K. Kinoshita and S.M. Reddy, "Cost-Effective Generation of Minimal Test Sets for Stuck-at Faults in Combinational Logic Circuits", IEEE Trans. on Computer-Aided Design, Dec. 1995, pp. 1496-1504.