A Test Synthesis Technique Using Redundant Register Transfers

Chris Papachristou
Computer Engineering Department
Case Western Reserve University
Cleveland, Ohio 44106

Mikhail Baklashov
Synopsys, Inc.
310 N. Mary Ave.
Sunnyvale, CA 94086

Abstract

This paper presents a test synthesis technique for behavioral descriptions. The technique is guided by two testability metrics which quantify the controllability and observability of behavioral variables and structural signals. The method is based on utilizing redundant register transfers in the datapath to produce a test behavior with better controllability and observability properties. This approach allows to avoid unnecessary insertions of test structures in the data path. A test scheme for conditional statements has been developed involving minimal changes in the controller. Our experimental results show improvements in fault coverage at modest hardware overhead.

1. Introduction

The need for fast time-to-market and increased productivity have motivated research towards high-level design and test (behavioral and register-transfer level (RTL)) [WagDey96]. Design for Test (DFT) techniques have been developed from ad hoc insertion of control or observe points to test structures such as scan chains and built-in self-test (BIST). Some EDA vendors have already automated DFT analysis and insertion for these test structures in their tools. The area of Test Synthesis has emerged dealing with the integration of DFT structures in a design.

Test synthesis at the behavioral and RTL has been the subject of considerable recent research with the aim to improve the testability of behavioral and RTL designs. Several researches addressed the issue of synthesizing testable datapaths with low area overhead. [ChPap91] presented a technique to synthesize testable datapaths starting from a behavioral description. [PaGuBr95] described a datapath allocation algorithm optimizing test registers to be used for BIST. In RaDePo94 a high level synthesis system based on partial scan selection was reported. The authors presented several techniques to improve testability of synthesized circuits including transformations, scheduling and assignment. [RaTy] presented a comprehensive test scheme for datapaths based on the Arithmetic BIST concept. [PaRu94] presented a testability analysis method at RTL using partial scan insertion along with VHDL behavioral modifications. Another approach on modifying behavioral descriptions was given in [ChKs94]. Based on this analysis, test statements which are executed only in test mode, are added to improve the controllability and observability of all the variables in the description. [PaCa95] proposed a general BIST scheme by adding test behavior, executed only in the test mode. A test behavior is obtained by inserting test points in the original behavior to enhance the testability of required internal signals. However, the test points need extra primary I/O implemented by TPGRs (Test Pattern Generator Registers) and MISRs (Multiple Input Signature Registers).

Depending on the area overhead and fault coverage constraints, the synthesized circuit may need post-RTL-synthesis testability analysis. For example, unnecessary extra TPGRs or MISRs in the datapath can be avoided by changing the test behavior. Such testability enhancements, if found, need to be accommodated in the controller behavior as well. Note that the hardware overhead related to such accommodations, if carefully chosen, will impact only the controller. This is the essential point in our approach as we will discuss it later, i.e. to effect post-RTL synthesis modifications without changing the datapath, possibly exploiting false control paths in test mode. In computation-intensive applications, such as digital signal processing (DSP), telecommunications and image processing the controller constitutes a small part of the total circuit and hence the controller hardware overhead for required testability modifications should be low.

In Section 2, we will review the background and fundamentals of our methodology. In section 3, we discuss testability analysis and metrics that we use. Sections 4 and 5 discuss our test synthesis approach whereas Section 6 highlights our test scheme for conditions. Section 7 provides experimental results, and Section 8 the conclusion.

2. Test Synthesis Approach

2.1 Behavioral Test Scheme

We consider the testability problem of RTL designs consisting of a pair of datapath-controller. The datapath is provided by a netlist of RTL components, i.e. functional blocks (FU) or ALUs and MUXes, and the controller is provided by a control flow FSM. These descriptions of datapath-controller are sometimes referred to as "functional RTL" and are usually modeled in VHDL or Verilog. A functional RTL design can be captured from a high level synthesis system or can be directly provided using a hardware design language e.g. VHDL. The behavior of the RTL design is embedded in the control flow FSM. We target computation-intensive applications such as DSP for which the controller implementation is relatively simple in comparison to the datapath.

Our approach to testing functional RTL designs is based on the notion of minimal behavioral BIST [PapCa95], [NonPap97]; for this scheme, all input/output variables and signals of the RTL are mapped into BIST input/output registers of the RTL circuit (TPGRs and MISRs). The scheme is minimal in the sense that no insertion is done within the RTL datapath; no internal variables are made BIST or test points. This means that the datapath is tested as a whole by placing a TPGR across all the primary inputs to the datapath, and an MISR across all the primary outputs. During testing, as the TPGR generates test patterns, the datapath is exercised according to its test mode behavior. Thus for the RTL design we have a design behavior performed on the datapath during normal operation along with a test behavior which is executed during testing. We distinguish accordingly the design behavior and the test behavior modes. A switch function between these two modes is incorporated in the controller.

Ideally, a test behavior should be the same as the design behavior; this would simplify the testing to testing the circuit according to this behavior for which it was synthesized. In reality, however, the RTL design is not synthesized with a priori testability considerations at the high level. Thus there are may be some testability problems in terms of signal controllability/observability at the RTL. Our aim is to modify the design behavior producing a test behavior so that the datapath will be successfully tested, in terms of fault coverage. Both the design and test behaviors are mapped into sequences that are specified in the controller.
of register-transfer paths at the RTL. However, the test behavior can utilize true as well as false register transfer paths, with the objective to improve testability. The identification of hard-to-test areas is based on the testability evaluation of VHDL behavioral descriptions. In our previous research, we have developed two testability metrics, randomness and transparency, which quantify controllability and observability of signals and variables are used. These metrics are reviewed later in this section after discussing the overall system.

2.2 System Overview

Shown in Fig. 1 are the main tools of our test analysis and synthesis technique. There are four major tools that we have implemented:

a) the VHDL compiler, b) the CDFG (control data flow) simulator, c) the register-transfer generator, and d) the test behavior generator. The fifth tool, the high-level synthesizer, is an independent tool which can be provided by other sources, commercial or not. The common interface format for all these tools is the VHDL language. The general flow follows. The VHDL descriptions are compiled into internal design by the VHDL compiler representation producing a CDFG graph. Our Graph Simulator performs testability analysis of the CDFG using probabilistic simulation. Several algorithms have been developed to compute the metrics employing the Monte Carlo Simulation; their description is beyond the scope of this paper. The High Level synthesizer transforms the CDFG into a functional RTL design, meaning an RTL datapath together with a control flow FSM (design behavior). The register transfer generator captures the register transfer set (true and false) from the synthesizer. Finally, the Test Behavior generator based on inputs from the Graph Simulator and the Register Transfer generator, produces a test behavior for the RTL circuit.

We have implemented the algorithms as well as a VHDL compiler in our tools. These tools are shown in Fig. 1 with gray color filled rectangular boxes.

2.3 Testability Metrics

Several testability metrics for RTL modules based on the notion of entropy have been introduced by researchers [ThAb89], [ChPap91]. These metrics are used to evaluate testability at an early design stage [ChLP92], [ChKS94], [Fl97]. These metrics quantify how the quality of pseudorandom values deteriorates as they propagate through the RTL circuit and how easily an erroneous value is propagated to an observable point.

In [ChPap91] the definitions for transparency and randomness were given only for a single FU. We extended those definitions to both the behavioral level and the RTL using a graph $G$ representing either CDFG (behavior) or a netlist of RTL modules (datapath). Fig. 2 shows a graph $G$ for these two metrics.

The transparency metric, $TG(v, e, G)$, of the input signal, $e$, regarding node $v$ in graph $G$ is defined as the probability of observing the one-bit data variations on signal $e$ through the graph outputs.

The randomness metric, $RG(v, G)$, of given node $v$ in graph $G$ is defined as the ratio of the node computed entropy and the node maximum entropy.

3. Testability Analysis Issues

3.1 Computation of Behavioral Metrics

During behavioral design capture phase the VHDL compiler verifies the syntax of the original VHDL description and produces a syntax graph. The syntax graph is a mapping of the language statements to nodes and edges according to the VHDL behavior. Since the syntax graph itself does not contain any information about data dependencies occurring in the graph we build the CDFG to allow for testability analysis.

The CDFG nodes represent input and output signal variables declared in the interface part of a design entity, process internal, and compiler generated intermediate variables. The randomness/transparency metrics of CDFG are computed for each edge/vertex in the behavior. The testability analysis proceeds to obtain the same values for the data path.

Monte Carlo simulation techniques have been proposed to simulate non-embedded modules and create comprehensive libraries of metrics to analyze the entire behaviors. The complexity of generating these libraries is very high. Unlike previous works, we perform computations of behavioral testability metrics over the entire VHDL behavior and completely rely on the whole CDFG Monte Carlo simulation techniques to compute the metrics. We calculate simulation-based controllability and observability metrics for VHDL behaviors represented in
terms of CDFG and develop algorithms for metric computation. Our behavioral testability model [BM97] prescribes for each node of the CDFG three attributes, one for controllability, and two for observability, transparency. The randomness computation algorithm is pretty straightforward and based on Monte Carlo Simulation of the whole graph. The transparency computations for the whole graph dictates many different considerations that must be taken into account. Development and implementation of the corresponding algorithm required a much more challenging effort as compared to randomness computation.

3.2 Relation of Behavioral and Structural Metrics

Structural testability metrics at the RTL can be computed based on the same Monte Carlo simulation process as with the behavioral metrics. However, the data path netlist is more complex than the CDFG for the computation because the allocation generates loops and Mux conditions due to resource binding. We propose another technique to compute structural RTL metrics from the behavioral that takes into consideration the results of hand-to-CDFG code into ALU to a data path. Let us consider a behavioral description of the circuit shown by the flow graph of Fig. 3 (a), where every edge of the graph is assigned a behavioral variable such as \( a \) and \( x \). Fig. 3 (b) shows an RTL implementation of the behavior where resource allocation binds the operations to two FUs. The interconnects of the RTL, e.g. \( w \), are produced by binding CDFG edges. In our test scheme when following the behavior, in every cycle through the behavior each CDFG edge is traversed by one pattern. However, in the RTL implementation of the behavior, the interconnects may be traversed by several patterns during the behavioral cycle, as many as the bound variables. Thus "blending" of the \( a \) and \( x \) patterns appears on the net \( w \) and these patterns are employed for testing \( FU_1 \). The testability metrics of the interconnect variables \( w \) in the RTL are computed by analyzing the probability distributions of the blended or bound variables. We exploit the special nature of binding or pattern blending to pre-compute the RTL testability metrics from the behavioral metrics.

Let \( R_a \) and \( R_x \) denote the randomness of \( a \) and \( x \), respectively, in the behavior of Fig. 3 (a). Let \( R_w \) denote the randomness of the interconnect \( w \) in the RTL structure of Fig. 3 (b). Further, let \( M_a \) denote the "moment" of variable \( a \), defined as follows:

\[
M_a \sum p_w \text{ and } M_x = \sum p_w \text{ where } \{p_w\} \text{ and } \{p_w\} \text{ are the probability distributions of variables } a \text{ and } x \text{ assuming that we apply a pseudorandom set of input patterns at the inputs of the behavior. Similarly } M_w = \sum p_w \text{ where } \{p_w\} \text{ is the probability distribution of the variable } w. \text{ Then}
\]

\[
R_w \approx C_1(R_a + R_x)/2 + C_2(M_a + M_x)/2 + C_3 \tag{3.1}
\]

This formula has been obtained using a simple analysis model and some approximation. The precise details are omitted here. As a result of this analysis, the randomness of the bound variable \( R_w \) can be expressed as a linear combination of the original \( a \) and \( x \) and the average of the behavioral variables that are bound into \( w \). The coefficients \( C_1 \), \( C_2 \), and \( C_3 \) depend on the behavior bit-length \( n \). Furthermore, for the transparency metric \( T_a \), \( T_x \) and \( T_w \) we have found a simpler relationship based only on averages:

\[
T_w = (T_a + T_x)/2 \tag{3.2}
\]

The details of the above derivation is in [BM97].

4. Test Synthesis Overview

After register allocation is done behavioral variables are bound or renamed by their corresponding registers. We identify all datapath transfers containing one destination register and two source registers connected to a functional unit (FU). However, only a few transfers are utilized in the behavioral path whereas a larger number of transfers are not used in the normal behavior. We distinguish respectively active and passive transfers. Passive transfers are non-active transfers by the design behavior but could be activated in a test mode; when that happens a passive transfer is also called a test transfer. Active transfers themselves are also largely used in test mode. However, the basic idea of this paper is to utilize some passive transfers during the test synthesis process with better testability properties.

The randomness of the destination register defines the randomness of that particular transfer. Fig. 4 shows a transfer in the behavioral path, active transfer \( A \), with low randomness, and two transfers, \( B_1 \) and \( B_2 \), with high randomness which are in the test paths (passive transfers). The test patterns appearing at Node 2 of \( A \) have low controllability to test successfully the adjacent path including multiplexer, FU and register. Since these transfers have higher controllability properties they may be considered as suppliers of test patterns for \( Reg_2 \) instead of the design transfer. However, test transfers may increase the number of test sessions and prolong the test application time. It will be shown later that, at most, only one additional test session is needed. Two test sessions can be scheduled in parallel.
the transfer to be the same because we want to keep exercising one path segment of the transfer from \( R(e) \) to \( A u l \). For exercising the other path segment from \( R(g) \) to \( A u l \), transfer \( B \), Fig. 4, in the second test path is needed, and \( B \) is left compatible to \( A \). Both compatible test transfers \( B_1 \) and \( B_2 \) need further scheduling to synthesize the complete test session. For example, if the first test transfer \( B_1 \) has primary input registers it can replace the design transfer without further scheduling. The second transfer will need to be scheduled and an additional test session will be created.

Let us assume that we found the first test transfer \( B_1 \) in Fig. 4 to be used as a replacement of the design transfer. Note \( B_1 \) is passive – not used by the design behavior. Then, consider the second test transfer \( B_2 \) which is active. We can omit the second transfer from the test behavior because we do not need to reexercise the same paths. We call such a register transfer \( B_2 \) equivalent to \( A \) transfer. Thus, when two transfers have the same source registers and share the same FU regardless of their destination registers they are said to be equivalent. Since equivalent transfers exercise the same paths, finding such equivalent transfers will allow to avoid additional test sessions. This will be shown in the next section.

Following is an overview of the major functions, steps S1 through S7, of our test synthesis approach.

- **S1. Rand Comp\(u\)ute** \((v, G)\) - Compute randomness and transparency in the CDFG graph.
- **S2. Bind Comp\(u\)ute** \((n, D)\) - Compute randomness and transparency in the datapath.
- **S3. R\(t\)tl Gen\(e\)rate** \((D)\) - Generate register transfer set \( S \) for the datapath.
- **S4. Re\(n\)ame** \((v, r, G)\) - Rename variables by their register names based on binding.
- **S5. Find\(\min\)** \((v, f, g, D)\) - Find a node corresponding to a minimal randomness transfer.
- **S6. Pool\(\rt\(l\)l** \((f, S)\) - From set \( S \) select the transfers with high randomness.
- **S7. Sched\(u\)ule** \((t, D)\) - Schedule an additional test session if needed.

The application of the functions will be demonstrated in the next section. Let us assume that during allocation one behavioral variable \( x \) with low randomness and another variable \( y \) with high randomness are mapped into the same register \( R(x, y) \). The randomness of \( R(x, y) \) increases because of combining the test patterns of each of the variables. There may also be a case when another two variables \( y \) and \( z \), with low enough randomness (but higher than the one of \( x \)), get mapped to a different register \( R(y, z) \) such that the randomness of \( R(y, z) \) remains low. The question that can be raised here is what variable to consider for potential testability problems. It is easily seen that variable with low randomness is of interest in this particular case rather than the variable \( x \) with the lowest randomness. Using this observation, we suggest to analyze the datapath randomness values to find the variable with relatively low randomness. If we start the analysis with the CDFG it may lead to obtaining wrong variables.

Once the datapath is synthesized all register transfers are extracted by analyzing the datapath netlist. When the minimum randomness value corresponding to a datapath register is found, we check the behavioral variables bound to that register. From the set of such variables we search for the variable with the lowest randomness. The variable with relative minimum randomness value is defined as:

\[ v_{\text{min}} = \min (S_r, r, D) \]

where \( S_r \) is a set of variables bound to register \( r \) of datapath \( D \). We will now use a formal notation for a register transfer.

A register transfer, \( d, v, d \) \( \equiv v, r_j, r_j \cdot a u, \) is an RTL transfer where \( d \) is a destination register, \( r_i \) and \( r_j \) are left and right source registers connected to arithmetic logic unit \( a u \), and \( v \) is the low randomness variable bound to \( d \).

### 5. First Case Example

The first case example is a third degree polynomial equation expressed by the formula \( y = (a \cdot x + b) \cdot x^2 + c \cdot x + d \). After parsing its VHDL description we generate a sequence of three address statements as follows:

\[
V_0 := x \cdot x; \quad V_1 := V_0 + b; \quad V_2 := x \cdot x; \quad V_3 := V_1 \cdot V_2; \quad V_4 := c \cdot x; \quad V_5 := V_4 + d; \quad y := V_3 + V_5
\]

The intermediate variables on the statement left hand sides are compiler generated. We used the high-level synthesis system SYNTEST [HarPa92] to obtain the scheduled data-flow graph and datapath.

After the scheduling and resource allocation phases each variable gets allocated into a register. Such variable-register pairs are shown for each node of the graph in Fig. 5. As seen, variable \( V_2 \) has the minimum randomness value, .66, and it gets binded into register \( r_2 \) when the datapath gets synthesized. The transfer corresponding to this randomness deterioration is \( r_2 \cdot x, r_1 \cdot r_4 \cdot a u, \). In Fig. 6, the randomness and transparency values are shown for each wire of the datapath. The register transfer segment having the minimum randomness value turns out to be the wire connecting the output of \( R(e) \) and the middle input of \( M(x) \). Binding specifications give us the only variable bound to this wire is \( V_2 \) and thus in this example it is the same variable for both, the graph and the datapath, that causes the randomness deterioration. As we go along we define a notation to be used in describing our technique. Let \( S^T \) be the set of all transfers found in the datapath. Let \( S^A \) be the set of active transfers used by the design behavior after register binding of variables is done; let \( S^P \) be the set of passive transfers in the datapath such that \( S^P \cap S^A = \emptyset \). Then \( S^T = S^A \cup S^P \). For this example \( S^A = \{ r_2 \cdot V_0, r_5 \cdot r_4 \cdot a u, r_3 \cdot V_3, r_6 \cdot r_2 \cdot a u, r_2 \cdot v, r_4 \cdot r_4 \cdot a u, r_1 \cdot V_1, r_4 \cdot r_7 \cdot a u, r_3 \cdot r_1 \cdot r_8 \cdot a u, r_3 \cdot r_1 \cdot r_3 \cdot a u, \}

\[ S^P = \{ r_1 \cdot r_4 \cdot a u, r_2 \cdot r_4 \cdot a u, r_1 \cdot r_2 \cdot a u, r_1 \cdot r_2 \cdot a u, r_2 \cdot r_3 \cdot a u, r_1 \cdot r_2 \cdot r_4 \cdot a u, r_2 \cdot r_4 \cdot a u, \}

Figure 5: DFG after register renaming of variables
For two transfer sets, for left and right source register compatibility, are register and one common source register with higher randomness. Each transfer carries control signal information that comprises opposed to some commutative operations it might perform. Each transfer carries control signal information that comprises multiplexer select and register load signals.

In this example, there are only three distinguished destination registers for 24 transfers, which are \( r_1, r_2, r_3 \). Therefore, if any transfer happens to have low randomness there may well be another compatible transfer (having the same destination register and one common source register) with higher randomness. To find higher randomness compatible transfers we check for two transfer sets, for left and right source register compatibility, respectively. For the transfer \( d_s \leftarrow r_i \cdot r_j \cdot alu \), the sets are

\[
K = \{ k : d_s \leftarrow r_i \cdot r_j \cdot alu \} \quad r_k \neq r_i
\]

\[
M = \{ m : d_s \leftarrow r_i \cdot r_m \cdot alu \} \quad r_m \neq r_i
\]

We do not consider the case when at least one of two sets is empty because the test behavior cannot be found according to our test technique. Note that in case an insertion of extra test point in the datapath is needed. The case when two sets are nonempty is of primary interest, since two compatible transfers for the test mode may be found to replace the one with low randomness. In this example, the transfer having the lowest randomness is \( A \), \( r_2 \leftarrow r_2 \cdot r_1 \cdot alu1 \). In the following we list the set of all possible transfers and particularly the sets \( K \) and \( M \) of transfers that are left and right compatible to \( A \). Again we need two transfers to replace \( A \) because we need maintain exercising the datapath elements corresponding to transfer \( A \). A preference is being given for transfers that have a primary input register as one of the source registers.

\[
r_1 \leftarrow r_2 \cdot r_3 \cdot alu1 \in K(\text{passive}, \in S^P) \quad (5.1)
\]

\[
r_2 \leftarrow r_3 \cdot r_5 \cdot alu1 \in K(\text{active}, \in S^A) \quad (5.2)
\]

\[
r_2 \leftarrow r_4 \cdot r_6 \cdot alu1 \in M(\text{active}, \in S^P) \quad (5.3)
\]

\[
r_2 \leftarrow r_4 \cdot r_7 \cdot alu1 \in M(\text{passive}, \in S^P) \quad (5.4)
\]

We now evaluate each of the above transfers. Transfer (5.1) has register \( r_2 \) as a destination register. This transfer could be activated with variable \( V_0 \) having higher randomness and having been scheduled two time steps prior to \( V_2 \). However, transfer (5.2) has a primary input register \( r_5 \) with the highest randomness. Transfers (5.3) and (5.4) also have the highest randomness but the preference is being given to the last one, (4). Thus, the second (5.2) and fourth (5.4) transfers have to be selected in test mode.

In the next step, before the test behavior is generated, we check for existing equivalent transfers in set \( S \). This allows us to remove redundancy in test behavior and hence prevent the datapath elements from multiple exercising. The idea is to swap one of the compatible transfers by an equivalent transfer which is active and hence it can remain the same in test mode. Fortunately, the previously selected transfers (5.2) and (5.4) have their equivalent transfers in \( S^T \). They are:

\[
r_1 \leftarrow r_5 \cdot r_4 \cdot alu1 \in S^P(\text{passive})
\]

\[
r_1 \leftarrow r_4 \cdot r_7 \cdot alu1 \in S^T(\text{active})
\]

We cannot tell apart these transfers as to their randomness quality because they all have primary input registers as source registers. This shows that there exist two choices to synthesize the test behavior. Note for test behavior synthesis the life times of variables mapped to the destination registers of two active transfers should not to be overlapped. Let us consider the first choice, when using the second (2) transfer. Suppose, that after scheduling and allocation variable \( V_0 \) mapped to the register \( r_2 \) of the second transfer is being read before the variable \( V_2 \) gets written into the same register. Therefore, the variable life times do not overlap and, in test mode, variable \( V_2 \) can be used in the transfer instead of variable \( V_0 \). The fourth (4) transfer can be eliminated due to the presence of its equivalent transfer. Now, we consider the second choice, when using the fourth (4) transfer. Note that we do not do the variable life time analysis because this transfer is passive. In the test mode, it becomes active with variable \( V_2 \) mapped to its destination register. The second (2) transfer is not needed because of existing its equivalent transfer in the design behavior. Finally, the results are two test behaviors shown in Fig. 7.

---

**Figure 6:** Behavioral Test Scheme for Polynomial

**Figure 7:** Modified Test Behaviors for Polynomial

In analyzing the first test behavior we see that in the test mode the operation, \( V_2 := x \cdot x \) is not used. It was replaced by the operation, \( V_2 := a \cdot x \), which is now mapped to the second (2) transfer. As seen in Fig. 8 the randomness values get considerably improved. The critical connection from the output of \( Reg^2 \) to the middle input of \( Mut^2 \) gets a randomness increase of .3. The only noticed decrease of .05 in the observability value at the left input of \( Mut^1 \) can be treated as an acceptable one.
6. Test Scheme for Conditionals

Our test synthesis approach based on test behavior cannot be directly applied to CDFGs with conditions. Conditional variables in the CDFG are not always explicitly defined relative to the internal variables of the algorithm. Disregarding conditions in many cases may lead to very low fault coverage results for synthesized implementations. We decided to cover the following two types of conditional structures. One which does not have a conditional variable in its branch basic blocks, and the other one which has a conditional variable in its branch basic blocks. The second type of conditional structure is of importance because of possible degradation in randomness and transparency of other variables in conditional branches. In the following we will describe briefly a specific technique to our test scheme for conditions.

There are two aspects of our technique for conditions: a) test controller implementation and b) compiler transformations. We want all conditions in test mode to be equally likely executed. This will provide the same probability of appearance of the test patterns in any conditional basic block and make possible to employ the randomness and transparency computations. If we try to randomize the status signals from the datapath we will allow for equally likely executing the conditional branches in test mode. Such randomizing can be treated as masking the results of conditional expressions during testing and does not depend on the number of conditions, or long conditions might be nested. A test logic comprising two logical gates that perform such randomizing has been designed. Its schematic is in Fig. 9 and can be considered as an insertion in the controller circuitry. Hardware overhead is negligible and includes two additional gates and a 1-bit TPGR. The next-state register is represented as a microprogram counter with CNT (count) and LD (load). N-condition selection bits select one of 2^N possible status signals. The selected signal is inverted, if needed, with condition inversion input and the appropriate conditional branch is executed.

In test mode the test signal is asserted allowing randomly to execute both branches. The condition test logic determines whether to assert CNT or LD based on the conditions that occurred in the behavioral mode. In the test condition mode asserting the signals CNT and LD would happen equally likely.

For the second aspect, we perform transformations on VHDL behavioral descriptions for IF-THEN-ELSE and CASE statements and insert additional statements which are executed only in test mode. Fig. 10 shows three transformations for THEN, ELSE and CASE branches.

When doing these transformations we assume that the high-level synthesizer will allocate the same statements in IF-THEN-

6.2 Experimental Results

The results of our test synthesis technique employing behavioral and structural testability metrics are in Table 1. The randomness and transparency computations for both, the behavioral-control-data-flow graphs and the structural datapaths have been done using our testability tools. All examples are 8-bit and have been synthesized using the high level synthesis tool, SYNTEST, and implemented with the computer-aided design tool, COMPASS. For fault simulation we used the AT&T's fault simulator, GENTEST. Random test patterns have been obtained by simulating TPGR in COMPASS. TPGRs, MISRs have been synthesized in the datapaths to

---

**Figure 8**: Modified Test Scheme for Polynom1

**Figure 9**: Condition Test Logic

**Figure 10**: Transformations for Conditions
compare their hardware overhead with that of the normal registers. For controller testing we wire the outputs of state flip-flops (FFs), which are load register and select multiplexer lines, to the primary outputs of the datapaths through specially designed multiplexers controlled by a 1-bit test signal. The table summarizes the complete hardware overhead for TPGRs and MISRs in the data paths, controller hardware overhead for handling the test statements, condition logic and additional multiplexers.

For the first two examples test behaviors have been found to get high enough fault coverage with insertion TPGRs and MISRs in the data path and modifying the controller description. The hardware overhead related to wiring control signals from FFs we counted as controller hardware overhead.

The results for FR feature our test synthesis technique applied to conditions. As it was described in the previous section, we found a specific test scheme along with the controller circuitry and obtained considerable improvement in fault coverage (18%) with hardware overhead 11%. The hardware overhead for the test condition logic is 0.4% and can be well neglected. For the last three benchmark examples the randomness and transparency values turned out to be very high and no test synthesis in terms of behavioral modifications has been performed. We applied only behavioral test scheme to test these three circuits and achieved high fault coverage with insertion only TPGRs and MISRs.

8. Conclusions

We have developed and tested a quantitative approach for behavioral and structural test synthesis using BIST. We proposed a testability technique to test the datapaths synthesized from behavioral descriptions with conditions. The basic idea behind the proposed methodology is to take advantage of redundancies inherent in the structure to change the test behavior. In analyzing redundant register transfers we found that by modifying the design behavior some of them might be used in test mode with improving the testability of the datapath. The behavioral modifications for conditional descriptions were developed along with the condition logic for the controller. All this required a change in the datapath controller. We accumulated such changes in controller descriptions and computed the hardware overhead. The advantages of our scheme are avoiding redundant hardware whenever possible and test controller simplicity. The experimental results showed improvement in fault coverage.

Although testability of the controller has not been dealt with specifically in this paper, our earlier approach on controller testability discussed in [PapCa5] could directly apply here as well. That approach uses structural analysis and insertion to enhance the testability of the controller and the interface between the datapath and the controller. Again, for DSP oriented applications, the overhead of the controller testability insertion, and the controller itself for that matter, in comparison to the datapath is relatively small.

### Table 1: Complete Test Results

<table>
<thead>
<tr>
<th>Circuit Name</th>
<th>Trans.Count (DP+Contr)</th>
<th>Fault Cover Before After Insertion</th>
<th>Test Time (cycles)</th>
<th>TPGR+MISR (8-bit)</th>
<th>Load, Sel FFs</th>
<th>Hardware Overhead (DP+Contr)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poly1</td>
<td>4934+4397</td>
<td>94.2% 96.3%</td>
<td>350</td>
<td>541</td>
<td>19</td>
<td>12.5%</td>
</tr>
<tr>
<td>Poly2</td>
<td>983+354</td>
<td>87.7% 98.0%</td>
<td>800</td>
<td>341</td>
<td>11</td>
<td>6.32%</td>
</tr>
<tr>
<td>Max</td>
<td>8413+2915</td>
<td>76.3% 94.3%</td>
<td>300</td>
<td>442</td>
<td>41</td>
<td>10.76%</td>
</tr>
<tr>
<td>Diff</td>
<td>6468+1697</td>
<td>92.4% 98.7%</td>
<td>250</td>
<td>443</td>
<td>27</td>
<td>8.37%</td>
</tr>
<tr>
<td>Elliptic</td>
<td>14547+9351</td>
<td>89.6% 97.3%</td>
<td>350</td>
<td>446</td>
<td>56</td>
<td>14.4%</td>
</tr>
<tr>
<td>Wave</td>
<td>5762+2387</td>
<td>92.5% 99.1%</td>
<td>400</td>
<td>1293</td>
<td>47</td>
<td>13.2%</td>
</tr>
</tbody>
</table>