On Automatic Generation of RTL Validation Test Benches Using Circuit Testing Techniques

Indradeep Ghosh† and Srivaths Ravi‡*
† Fujitsu Laboratories of America, Sunnyvale, CA 94086
‡ NEC Laboratories America, Princeton, NJ 08540

ABSTRACT

In this paper, we examine how good validation test benches can be automatically generated starting from the RTL description of a circuit. We develop our methodology based on extensive experiments performed with several popular benchmarks as well as industrial circuits. We try to leverage off a large body of work in the field of circuit testing and study how test sets derived for catching manufacturing defects fare from the standpoint of design validation. For this purpose, we perform an extensive empirical study using stuck-at test sets from gate-level implementations synthesized under various constraints. The experiments demonstrate that a good logic-level stuck-at test set is also an excellent RTL validation test bench. However, since we are dealing with RTL designs here and sequential logic-level ATPG is an expensive algorithm, we devise some methods to obtain good quality validation test vectors directly at the RTL. We use these results to enhance an existing RTL ATPG tool and show that test benches that can achieve good logic-level fault coverage and thus design validation coverage can be derived in our framework.

Categories and Subject Descriptors

B.0 [**Hardware**]: General; B.2.3 [**Hardware**]: Arithmetic and Logic Structures- *Reliability, Testing and Fault-Tolerance*; B.7.3 [**Hardware**]: Integrated Circuits- *Reliability and Testing*; B.8.1 [**Hardware**]: Performance and Reliability- *Reliability, Testing and Fault-tolerance*

General Terms

Experimentation, Reliability, Verification

Keywords

Validation, Design Validation, Coverage Metrics, Code Coverage, OCCOM, Branch Coverage, Path Coverage, Toggle Coverage, Testing, ATPG, Test Generation, Fault Coverage, RTL testing, RTL ATPG, Test Sets, Testbench, Universal Test Sets

1. INTRODUCTION

With the increasing design complexities and time-to-market pressure, a large number of IC chips today are being designed at the RTL. This has led to the emergence of a large number of synthesis tools at the RTL. The design of a chip at the RTL brings with it additional burdens of validation and verification, area/delay/power estimation, *etc* as well as the task of interfacing with the lower levels of the design hierarchy. Automating these design flow steps is necessary for sustaining design effort at the higher levels of the design hierarchy.

*This work was done when the author was a summer intern at Fujitsu Labs of America.

GLSVLSI'03, April 28–29, 2003, Washington, DC, USA.

For verification purposes, simulation-based techniques (increasingly at the RTL) are mostly used in the chip manufacturing industry. In this process, the circuit to be verified is simulated with a set of test vectors which is popularly known as a test bench. The output responses of the circuit are captured and compared for correctness with expected outputs derived from the specification of the circuit. The thoroughness of the verification process depends on the ability of the test bench to exercise the design completely. If portions of the design remain unexercised, the probability of bugs existing in those regions increases drastically. In the industry, the most common practice is to craft test benches by hand that can cover a large percentage of design in the RTL description. In the process, the designer would like the test bench to stress the design in a realistic manner and approach corner-cases, whenever possible. Test benches that yield high coverage on RTL coverage metrics such as statement, branch, condition, toggle, OCCOM (observability based code coverage) *etc* are considered good test benches. While high coverage numbers are not sufficient to cover all aspects of design verification, they provide a quantifiable means for a designer to evaluate and compare test benches. A good verification test bench should at least have close to 100% coverage for the traditional coverage metrics like statement, branch, and condition. In this paper, techniques for automating this test bench writing process are presented. Note that while automatically deriving test benches at the RTL, there is a risk of validating a buggy circuit. Therefore, in this work, we assume that a golden RTL model is present from where the test benches may be derived. If a golden RTL circuit is not available, the test vectors obtained from an implementation can be applied to its executable specification and may produce different outputs when the bugs are excited in the implementation. Further, this technique may be used to validate the logic-level circuit derived from an RTL description. Thus, automatically generated validation test benches at the RTL can aid the verification process to a large extent.

In this work, we address the above problem by first investigating the following question: "*Do test vectors generated for detecting manufacturing faults in a gate-level netlist corresponding to the RTL description serve as good validation test benches?*". If the answer to this question is yes, we can reduce the validation test bench generation problem to an automatic test pattern generation (ATPG) problem. However, this does not simplify the problem a great deal since the most popular method of performing sequential ATPG is at the logic-level, which is computationally expensive. Also the logic level design may not be available at an earlier point in the design process. Finally, the test vectors generated from the logic-level circuit may validate the bugs already present there. However, recent success with RTL ATPG techniques indicate that the ATPG problem can be tackled efficiently at the RTL because of the advantages of lower complexity, smaller design size and an understanding of the functionality. Typically, an RTL circuit will have fewer number of primitive elements in its circuit graph than its corresponding logic-level circuit. Hence, the problem size for performing ATPG is reduced thus reducing the ATPG complexity. Therefore, we examine the ATPG problem at the RTL and see how current RTL testing techniques can be augmented to produce good validation testbenches.

The paper is organized as follows. Section 2 presents some previous work in this line of study. Section 3 details the experimental setup. Section 4 investigates the performance of logic-level test benches at the RTL. Section 5 presents techniques to enhance the logic-level stuck-at coverage of a test bench derived at the RTL. Section 6 presents our integrated RTL validation/ATPG environment as well as results of its application on several industrial benchmarks, while Section 7 concludes.

2. PREVIOUS WORK

In this section, we examine related work relevant to this paper from the

areas of design validation, RTL ATPG and fault modeling.

A. Coverage metrics for HDL designs

Popular RTL coverage metrics are mostly borrowed from software testing techniques [2]. In software testing, coverage metrics are based on activation of statements, branches or sequence of statements due to a given set of program stimuli[3]. Such line or branch or path coverage metrics are now used in commercial tools performing design validation[4].

However, these coverage metrics are inadequate in design validation since they focus only on controllability (statement activations) and not on observability (transmission of possible errors to the circuit outputs). Recently, an observability-based statement coverage metric OCCOM[10] was proposed that enhances the standard statement coverage metric by incorporating observability criteria. The OCCOM metric is based on using the concept of a *tag* to model the possibility of an error at a location in the RTL design. Given a vector and a location in the RTL code, a tag at the location is said to be covered by the vector under OCCOM if it is determined that the tag can be propagated to some output by the vector.

B.RTL ATPG and fault modeling

There has been numerous attempts to generate test vectors at the RTL targeting logic-level stuck at faults as well as various RTL modeled faults [5, 6, 7, 8, 9].

Hierarchical functional HDL circuits have been targeted for test generation in [5]. However, the algorithm explicitly targets stuck-at faults at the logic level and there is no explicit RTL fault model used. Each RTL module is tested with a precomputed stuck-at test set from the primary inputs of the RTL circuit. Note that testing of RTL modules using precomputed test sets was first introduced in [6]. Similar philosophy is used in [7] but the test path generation algorithm makes use of regular expression based analysis. The above RTL techniques require that the precomputed test sets of RTL modules must be stored in a test set library for every bit-width and every implementation of a module possible. Since it is almost impossible to predict a logic implementation of a module in a constraint-driven logic synthesis scenario where the module implementations are essentially technology-mapped boolean equations, the above methods can be approximate. This paper tries to address this issue in Section 6.

Table 1: Benchmark Characteristics

CKT	#Lin.	#Proc.	Type	# Gates	# FFs
B01	110	1	flat	47	5
B02	70	1	flat	29	4
B08	89	1	flat	168	21
B14	509	1	flat	4776	245
Paulin	130	1	flat	39558	227
GPIO	1002	20	hierarchical	1720	148
ATMS	3214	84	hierarchical	8160	1490
MEMX	10674	651	hierarchical	16871	1954

Table 2: Performance (manufacturing test and validation coverages) of logic-level stuck-at test sets at the RTL for area-optimized circuits

CKT	FltC (%)	StmC (%)	BrC (%)	CondC (%)	TogC (%)	OCCOM (%)
B01	100.0	100.0	100.0	100.0	100.0	100.0
B02	100.0	100.0	100.0	100.0	100.0	100.0
B08	99.5	100.0	100.0	100.0	100.0	97.9
B14	95.1	100.0	100.0	99.1	100.0	92.5
Paulin	99.7	100.0	100.0	100.0	100.0	100.0
GPIO	99.4	100.0	100.0	100.0	100.0	100.0
ATMS	95.4	100.0	100.0	99.1	100.0	93.8
MEMX	95.0	100.0	99.8	98.9	100.0	92.1

Other ATPG techniques such as [8], [9] define new fault models at the RTL and generate tests according to that model. They then validate that coverage by doing logic-level fault simulation and show close correlation to the RTL model. In both these cases, a bit-error model is assumed in which each bit in each variable at the RTL is injected with a stuck-at 0 or stuck-at 1 fault. In addition to that, a stuck-at true/false fault is assumed for each condition at the RTL[8]. However, since the focus of the work is exclusively on ATPG, no attempts are made to correlate ATPG and validation coverage metrics. The work in [11] on RTL fault modeling is very pertinent to this paper. In this work, the author defines a new coverage metric called validation vector grade (VVG) at the RTL and shows that this coverage metric closely tracks fault-coverage at the logic level. The VVG metric is actually a variation of the bit-error model discussed above. In addition to bit stuck-at faults for all variables, it also injects faults at all fanout branches much like logic-level fault simulation.

3. EXPERIMENTAL SETUP

The experiments are done on eight RTL circuits out of which four are taken from the ITC 99 benchmark suite [13]. Of these, *B1* is an FSM that compares serial flows, *B2* is an FSM that recognizes BCD numbers, *B8* is a circuit that finds inclusions in a sequence of numbers, and *B14* is a microprocessor that implements an instruction set that is a subset of the Viper microprocessor instruction set. The fifth example *Paulin* is a popular filter benchmark from academia. The last three examples are industrial circuits from real life designs in Fujitsu. *GPIO* is a general purpose input/output controller. *ATMS* is a part of an ATM switch, and *MEMX* is a part of a memory controller. The characteristics of the circuits are summarized in Table 1. Columns 2 and 3 indicate the number of lines and processes, respectively, in the RTL description (VHDL) of the different circuits, while Column 4 indicates the RTL design style (flat/hierarchical). Columns 5 and 6 capture the post-synthesis gate-level statistics in terms of the number of gates and flip-flops, respectively. The RTL size of the examples vary from 70 lines to more than 10,000 lines, while the logic-level sizes vary from 30 gates to as high as 40,000 gates. These numbers were obtained using the area optimized scripts in Synopsys Design Compiler for Fujitsu's 0.25 micron standard cell library. The logic netlist characteristics for other synthesis scripts are a little different.

Table 3: Performance (manufacturing test and validation coverages) of logic-level stuck-at test sets at the RTL for delay-optimized circuits

CKT	FltC (%)	StmC (%)	BrC (%)	CondC (%)	TogC (%)	OCCOM (%)
B01	100.0	100.0	100.0	100.0	100.0	100.0
B02	100.0	100.0	100.0	100.0	100.0	100.0
B08	99.4	100.0	100.0	100.0	100.0	91.5
B14	94.9	100.0	100.0	98.2	100.0	91.9
Paulin	99.2	100.0	100.0	100.0	100.0	100.0
GPIO	99.2	100.0	100.0	100.0	100.0	98.1
ATMS	94.5	100.0	99.4	98.6	100.0	92.1
MEMX	93.1	99.9	99.2	98.1	100.0	90.5

Table 4: Performance (manufacturing test and validation coverages) of logic-level stuck-at test sets at the RTL for mixed-optimized circuits

CKT	FltC (%)	StmC (%)	BrC (%)	CondC (%)	TogC (%)	OCCOM (%)
B01	100.0	100.0	100.0	100.0	100.0	100.0
B02	100.0	100.0	100.0	100.0	100.0	100.0
B08	99.4	100.0	100.0	100.0	100.0	91.5
B14	95.0	100.0	100.0	98.2	100.0	91.9
Paulin	99.5	100.0	100.0	100.0	100.0	100.0
GPIO	99.4	100.0	100.0	100.0	100.0	99.8
ATMS	95.3	100.0	99.5	99.0	100.0	92.4
MEMX	93.4	100.0	99.4	98.6	100.0	91.6

4. RTL VALIDATION AND STUCK-AT TESTS

In this section, the performance of the logic-level stuck-at coverage metric is evaluated at the RTL. This is done in the following way. First, a good quality logic-level stuck-at test set is obtained for a circuit by running HITEC and STRATEGATE on the the logic-level circuit and taking the union of the test sets. Purposefully those circuits are chosen for which the stuck-at fault

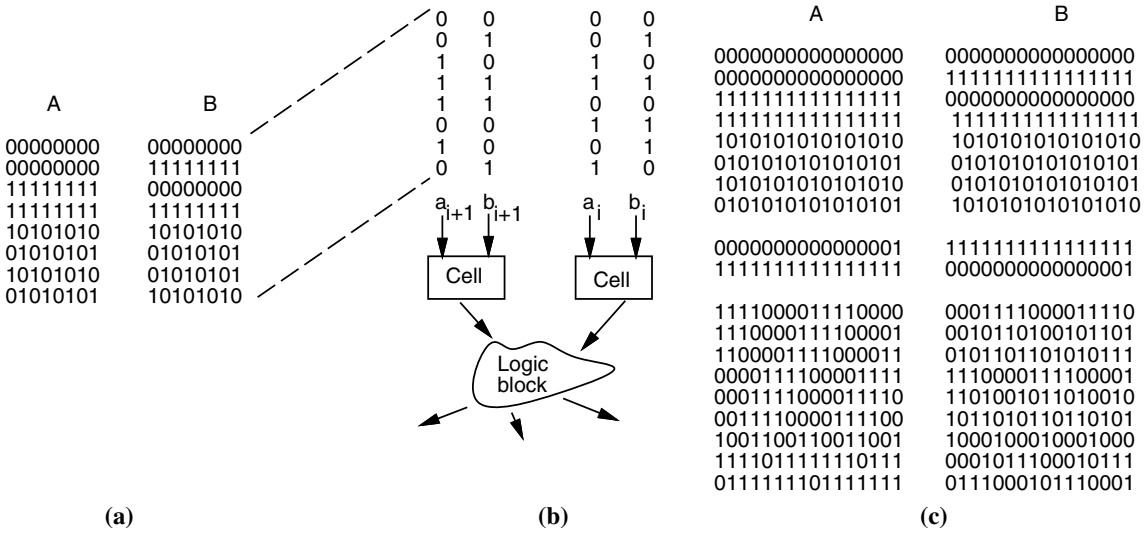

Figure 1: 8-bit generic test set and comprehensive test set of a 16-bit adder

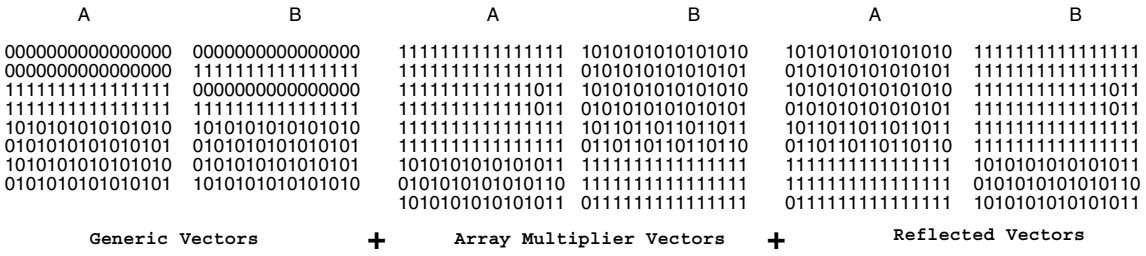

Figure 2: Comprehensive test set derived for a 16-bit multiplier

coverages are greater than 95%. Then, the test set is fed into TransEDA Verification Navigator to generate the traditional RTL coverage numbers. Finally, the inhouse coverage analysis tool is run to obtain the OCCOM numbers. The experiments are first done with a synthesis script that results in area-optimized circuits. In that script, infinite clock period is given to the synthesis tool but with a tight area constraint. These results are shown in Table 2. The experiments are repeated with delay-optimized circuits where the script parameters are reversed. These results are shown in Table 3. Another set of experiments are done with circuits, where, the synthesis script equally considers area and delay constraints to obtain a logic circuit somewhere between the two extreme design points of the previous cases. These circuits are termed mixed-optimized circuits. The results for the mixed optimized circuits are shown in Table 4. In all cases, Synopsys Design Compiler is used for logic synthesis.

In the experiments, the following traditional coverage metrics are considered - statement, branch, condition and toggle (Columns 3, 4, 5 and 6, respectively, in the tables). Path coverage is not used as the coverage analysis tools cannot handle complete path coverage analysis (since there are exponential number of paths). For condition coverage, the *focussed expression coverage* metric is used. This metric only requires the care vectors for each clause of the expression thus eliminating the need for impossible vectors at the expression.

From the tables, it is clear that for all types of circuits if the fault coverage is greater than 98% at the logic level (Column 2) then it results in 100% coverage for all traditional coverage metrics at the RTL. In case of OCCOM coverage (Column 7), the coverage numbers are also quite high but not perfect. This is because the OCCOM coverage analysis is inherently pessimistic. This pessimism is due to the limited attributes attached to a tag. A tag has only a sign attribute attached to it ($+\Delta$ or $-\Delta$) to determine if it can be propagated forward.

From the above experiments three important conclusions can be drawn:
Conclusion 1: A good stuck-at test set at the logic level is also a good validation test bench at the RTL in terms of RTL coverage metrics.
Conclusion 2: The testability properties of a logic circuit does not vary too much due to the RTL synthesis script used and the resulting logic-level test set on any implementation is a good RTL validation test set. This observation is in agreement with the one made in [12].
Conclusion 3: *From conclusions 1 and 2 it follows that if a test set can be generated directly from an RTL circuit targeting high coverage of logic-level stuck-at faults for some logic-level implementation of the RTL circuit, then that test set will be good for RTL validation too.*

In this section, it has been shown that a good logic-level stuck-at test set is also a good validation test bench. However, the aim here is to generate good validation test benches directly at the RTL. Hence, some techniques are required to obtain good logic-level stuck at test sets directly from the RTL circuit without going through complete synthesis and then using a logic-level ATPG tool. Such techniques are discussed in the next section.

5. GENERATING RTL TEST BENCHES

A technique for ATPG at the RTL targeting stuck-at faults is presented in [5]. If this technique is to work well for any logic-level implementation of an RTL circuit, then two problems need to be tackled. They are the presence of arithmetic modules and random logic blocks. A preliminary approach for testing realization-independent blocks in a design subject to a behavioral fault model is given in [19]. However, this will not work for the stuck-at fault model. Arithmetic modules are synthesized into logic implementations which depend on the synthesis script. Thus, if a test generated at the RTL for an arithmetic operation is to provide good stuck-at fault coverage at the logic level irrespective of implementation, it is essential to find a set of comprehensive test vectors that work reasonably well for any implementation of the operation. This is discussed next.

5.1 Handling RTL arithmetic modules

Only those arithmetic modules need to be tackled that are synthesized as atomic operations by RTL synthesis tools like Synopsys Design Compiler. If an arithmetic module at the RTL is designed with smaller logic components, then each of those components can be tested separately without considering

the arithmetic module as a whole. These synthesizable atomic operations can be broadly classified into four categories - adders/subtracters, multipliers, comparators, and shifters/rotators, which are tackled next.

5.1.1 Adders/Subtracters

First, a set of generic test vectors are described. Consider the vector set in Figure 1(a). It consists of eight vectors and these vectors are usually very good for detecting a large number of faults at the logic level for many arithmetic modules. The intuition behind this is that most arithmetic modules are composed of an array of two-bit cells as shown in Figure 1(b). These vectors provide all symmetric vectors to two consecutive cells in the array.

Table 5: Performance of comprehensive test set on adders (19 vectors)

BW	Area Optimized		Delay Optimized		Mixed Optimized	
	#Gates	FltC (%)	#Gates	FltC (%)	#Gates	FltC (%)
4	130	100.0	193	100.0	160	100.0
8	212	100.0	590	95.7	408	99.6
16	468	100.0	1457	95.2	899	99.8
32	980	100.0	2824	95.4	1935	99.7

A comprehensive test set for the adder is obtained by combining the generic vectors shown above with the test sets for a slow implementation and a fast implementation. A ripple carry adder is used as the slow implementation. This is *C-testable* with 8 vectors for any bit-width. A carry-in input to the least significant bit of the adder is not assumed here. However, the property holds in that case too with a small modification to the least significant bit in the test set. Only test vectors that are not already present in the generic test set are added. On top of this, a test set comprising of 12 vectors is derived for a 4-bit blocked carry lookahead adder used popularly in the industry. The non-overlapping vectors of this test set is combined with the above test set to obtain the comprehensive test set of an adder which comprises of 19 vectors. This is shown in Figure 1(c). Note that all the vectors in the comprehensive test set follow a pattern. Thus they can be derived during test generation for any bit-width adder. The last 9 vectors in the test set may not seem to follow a pattern at first look but if 4-bit chunks of the test vectors are examined then the pattern will be evident. On fault simulating different adders of various implementations with the test set good fault coverage is obtained in all the cases. The results are summarized in Table 5. Column 1 indicates the bit-width of the adder. Columns 2 to 7 indicate the gate count and fault coverages for different versions of the logic-level implementation (area optimized, delay optimized and mixed optimized). The results show that the test set derived above may be used as a comprehensive test set during RTL test generation of addition operations.

Table 6: Performance of comprehensive test set on multipliers (26 vectors)

BW	Area Optimized		Delay Optimized		Mixed Optimized	
	#Gates	FltC (%)	#Gates	FltC (%)	#Gates	FltC (%)
4	198	98.5	250	97.5	233	97.7
8	1052	97.2	1091	96.3	1080	96.7
16	4682	96.5	4810	95.5	4709	96.3
32	19668	96.4	19951	95.1	19707	96.3

Subtracters are nothing but 2s-compliment adders. In that case the operand B is complimented and the carry-in to the least significant bit of the adder is set to 1. A comprehensive test for a subtracter can be obtained by changing the adder test set to take into account the above modifications. The changed test set will again feed the comprehensive adder test to the adder part of the subtracter. On fault simulating this test set on different subtracters, similar results are obtained as above.

5.1.2 Multipliers

The comprehensive test set of a multiplier is shown in Figure 2. It consists of the eight generic vectors and an array multiplier test set taken from [17]. The array multiplier test set follows a pattern and can be used to generate test sets for multipliers of any bit-width. These together constitute 17 vectors. The last 9 test vectors are obtained my interchanging the ports of the array multiplier test set to make the test set symmetrical. The performance of these vectors on different multiplier implementations are summarized in Table 6.

A	B
00000000	00000000
11111111	11111111
00000000	00000001
00000000	00000010
00000000	00000100
00000000	00001000
00000000	00010000
00000000	00100000
00000000	01000000
00000000	10000000
00000001	00000000
00000010	00000000
00000100	00000000
00001000	00000000
00010000	00000000
00100000	00000000
01000000	00000000
10000000	00000000

(a) A == B

A	B
00000000	00000000
11111111	11111111
11111110	11111111
00000000	00000001
00000000	00000010
00000000	00000100
00000000	00001000
00000000	00010000
00000000	00100000
00000000	01000000
00000000	10000000
00000010	00000001
00000100	00000011
00001000	00000111
00010000	00001111
00100000	00011111
01000000	00111111
10000000	01111111

(b) A < B

Figure 3: Comprehensive test sets for 8-bit equal-to and less-than comparators

Again, good fault coverage is obtained for all the implementations using this comprehensive test set.

5.1.3 Comparators

Two type of comparators are tackled here - the equal-to and the lesser than. All other comparators can be derived from these two comparators by switching operands or complimenting the output or both. Figure 3(a) shows the test set for a simple *XNOR-AND* equal-to comparator. This test set has a length of (2n+2) where n is the bit-width of the comparator. Thus, it is linearly growing with size but still can be derived at runtime. The generic vectors are not good for comparator testing. Thus the comprehensive test set for an equal-to comparator just consists of this (2n+2) vectors. In Table 7, its performance is shown for different versions of the comparator. Note that a blank square in the table means that Design Compiler has not been able to generate any other solution with the cell library that has been used. It is evident from the table that the vectors actually provide perfect coverage for all equal-to comparators.

Table 7: Performance of comprehensive test set on equal-to comparators (2n+2 vectors)

BW	Area Optimized		Delay Optimized		Mixed Optimized	
	#Gates	FltC (%)	#Gates	FltC (%)	#Gates	FltC (%)
4	39	100.0	77	100.0	-	-
8	79	100.0	-	-	-	-
16	193	100.0	179	100.0	173	100.0
32	405	100.0	629	100.0	491	100.0

For the less-than comparator also a simple implementation test set is used as the comprehensive test set. This is shown in Figure 3(b) and it consists of (2n+2) vectors. Again the test vectors follow a pattern as in the previous cases. The fault simulation results are provided in Table 8 and shows that the comprehensive test set is quite good.

Table 8: Performance of comprehensive test set on less-than comparators (2n+2 vectors)

BW	Area Optimized		Delay Optimized		Mixed Optimized	
	#Gates	FltC (%)	#Gates	FltC (%)	#Gates	FltC (%)
4	73	100.0	105	95.3	101	100.0
8	172	100.0	305	95.0	263	100.0
16	386	100.0	719	95.3	595	100.0
32	760	100.0	1099	96.4	920	97.9

5.1.4 Shifters and Rotators

Shifters of the form shift(I,S) where I and S are both signals or variables are usually not supported in a synthesis tool. However, Figure 4.a. shows

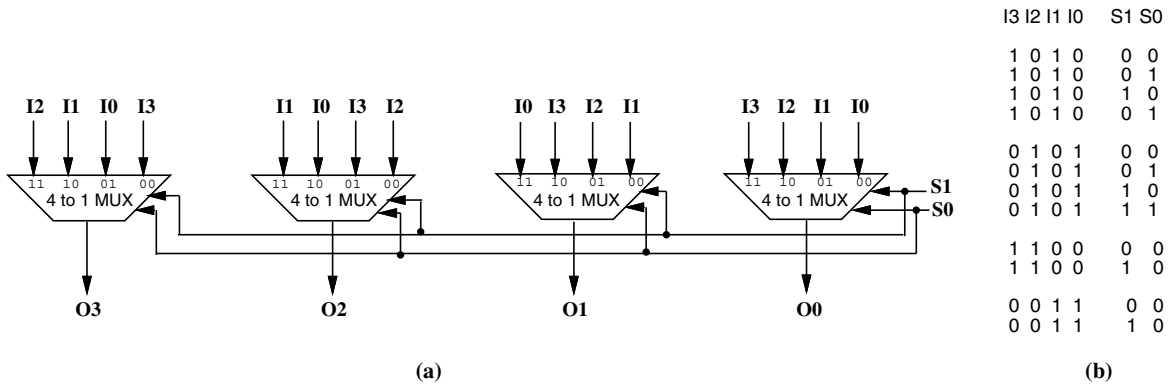

Figure 4: 4-bit generic rotator

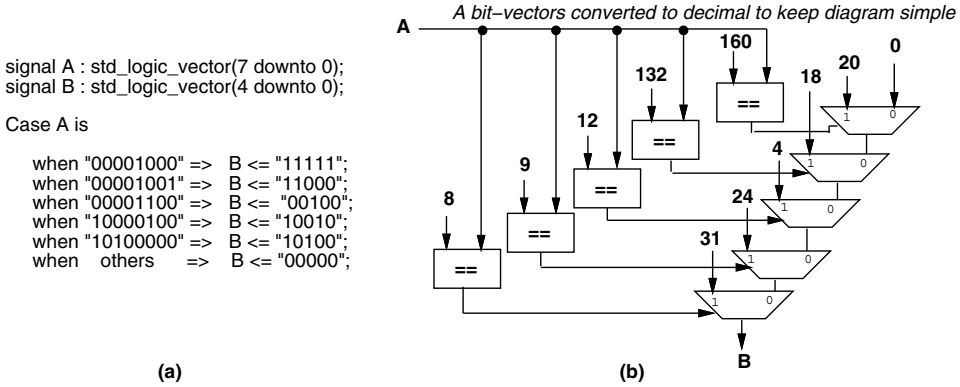

Figure 5: Random logic HDL code and its crude implementation

a generic implementation of a 4-bit right-rotator. Here variable S needs to be of size $log_2(width_of(I))$. By changing the input lines appropriately or by placing 1s or 0s at certain input lines any type of shifter or rotator can be obtained. The rotator of Figure 4(a) has a well defined test set of length *4n -4*, *i.e.* 12 vectors. This test set is shown in Figure 4(b). It results in 100% logic stuck at coverage and follows a pattern as before. When this test set is used on a 4-bit right shifter of similar architecture, the fault coverage is 91% due to the swept constants. Unlike the previous operations further experiments cannot be performed as Design Compiler allows shifts by constant numbers only.

5.2 Handling RTL random logic blocks

Random logic blocks in the RTL arise from *Case* statements or nested *if-then-else* blocks. In this case also, logic synthesis is done based on a truth table and the design constraints. The final logic-level implementation is unclear at the RTL. Hence, we need to formulate a systematic procedure to obtain a comprehensive test set for these block. This procedure is discussed next.

Consider the *Case* statement in Figure 5(a). It constitutes an address decoder. In this case the first thing to do at the RTL is to cover all the cases with the test vectors. Note that only exciting each case with different addresses is not enough. The effect of the decoded address (B) should also be propagated to the primary outputs. The algorithm presented in [5] is used for this purpose. This can be termed as observability enhanced code coverage. Using the OCCOM metric will generate similar results. When the resulting vectors are fault simulated at the logic level for different synthesis scripts, the fault coverage obtained is not very good. This will be evident from the data presented below.

In order to increase the coverage, one extra step needs to be performed at the RTL. First, a crude logic-level implementation in generated for the *Case* statement. This is a straight forward priority encoder implementation and is shown in Figure 5(b). The constants in the circuit are swept away and then this crude logic implementation is fed to a logic-level ATPG tool to obtain a crude implementation test set. Since these random logic blocks are usually not very big, the logic ATPG is not a very expensive operation. This test set is appended with the earlier test set to obtain a comprehensive test set for the random logic block directly at the RTL. From the experimental data, it will be evident that the performance of this test set is reasonably good for different logic implementations of random logic blocks.

The experimental data is provided in Table 9. Five random logic generating *Case* statements are extracted from the industrial RTL circuits. They are synthesized to logic using the three different synthesis scripts described earlier. The circuits are fault simulated using PROOFS first with the earlier (only observability enhanced statement coverage) test set and then the enhanced test set (adding crude implementation test set). For generating the crude implementation test set, HITEC is used on the crude logic implementation of the case statement. It can be seen from the table that the observability enhanced code coverage test set usually provides a fault coverage of around 70% which is clearly not enough. However, the enhanced test set by combining the two methods does provide greater than 90% fault coverage for all cases and in most cases the coverage is greater than 95%. Hence by using the above method, random logic blocks can be tested directly at the RTL for good fault coverage at the logic level.

6. ENHANCEMENTS TO RTL ATPG

In this section, the insights gained in the previous section are incorporated into the RTL ATPG tool of [5]. The previous version used a test set library for RTL modules with various implementations and bit-widths. This library has obviously been incomplete and used non-universal test sets. If the implementation of an RTL module is not clear, then the test set for some arbitrary implementation has to be used. As a result fault coverage of arithmetic modules have suffered. Also, the random logic blocks have been tested with observability enhanced code coverage resulting in moderate fault coverage.

In the new version, the universal test-sets obtained in Section 6 are used. This results in the removal of the test set library as the tests for all arithmetic modules are made to follow a pattern and, thus, can be derived at runtime. Also, the crude implementation test sets are used for the random logic blocks by using a logic ATPG tool to derive the stuck-at test sets for those small portions of logic and then justifying and propagating them at the RTL. The test sets were then fault simulated at the logic level using PROOFS [18] and the fault coverage noted. The results are summarized in Table 10. In each case,

Table 9: Performance of RTL tests on Random logic blocks

Block	Area Optimized			Delay Optimized			Mixed Optimized		
	# Gates	Earlier	Enhanced	# Gates	Earlier	Enhanced	# Gates	Earlier	Enhanced
C1	84	75.4	97.2	96	72.7	96.5	87	75.8	97.1
C2	108	73.2	96.8	134	70.8	96.2	121	71.4	96.6
C3	259	73.0	95.8	290	70.4	93.6	265	71.0	95.1
C4	451	71.2	93.6	501	67.3	90.5	475	69.6	92.1
C5	564	68.5	91.4	632	65.5	90.1	599	66.2	90.4

Table 10: Fault coverage improvements by enhancing the RTL ATPG tool [5]

CKT	Orig		Enhanced		Logic ATPG	
	FltC (%)	Time (sec)	FltC (%)	Time (sec)	FltC (%)	Time (sec)
B01	99.3	0.2	100.0	0.2	100.0	0.4
B02	98.7	0.3	100.0	0.3	100.0	1.4
B08	96.5	0.3	98.2	0.3	99.5	2.1
B14	94.3	10.5	95.2	11.2	95.0	23.8
Paulin	99.1	5.2	99.4	5.2	99.2	154987.2
GPIO	97.6	65.2	99.5	78.1	99.4	55.6
ATMS	93.1	2032.2	95.4	2156.9	95.1	59657.2
MEMX	91.5	4056.1	94.2	5960.0	93.8	80701.6

the average results are reported from using three different implementations of the RTL circuit. The CPU time is for a Sparc Ultra60 with 512MB of memory. It can be observed that in most cases, the implementation of the above techniques in the RTL ATPG tool results in higher logic-level fault coverage for the circuits. In fact, the fault coverages for even highly control flow intensive circuits where the Gates/(RTL lines) ratio is low, the RTL ATPG fault coverage is very close to the logic-level ATPG fault coverage. Also the CPU time required for RTL ATPG is much lower than logic-level ATPG for all the large circuits. Note that the CPU time is also averaged over the three implementations. Thus, the RTL ATPG algorithm can be used to generate good validation test vectors at the RTL.

7. CONCLUSIONS

In this paper, the correlation between RTL and logic-level coverage metrics is investigated. Some insights are obtained on the different types of coverage metrics and their effectiveness in detecting faults and errors. The experiments point out the fact that any good logic-level stuck-at test set is also a very good validation test set at the RTL. Thus, if a test set can be generated at the RTL that provides good logic-level stuck-at coverage then it can be used as a good validation test set during RTL verification. Some techniques are explored to handle large synthesizable modules and random logic blocks at the RTL and obtain good stuck-at test sets for them directly at the RTL. These techniques and insights are incorporated into an RTL ATPG tool and improvements in fault coverage numbers are demonstrated. The RTL ATPG algorithm is much faster than the logic ATPG algorithm for larger circuits and with the above improvements the fault coverage numbers for both the cases are comparable. In fact, the RTL ATPG does better for some circuits. Thus, the improved tool can be used to generate good validation test vectors at the RTL.

8. REFERENCES

[1] M. Abramovici, M.A. Breuer, and A.D. Friedman, *Digital Systems Testing and Testable Design*, IEEE Press, New York, 1990.

[2] B. Beizer, *Software Testing Techniques*, Van Nostrand Rheinhold, New York, second edition, 1990.

[3] B. Marick, *The Craft of Software Testing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

[4] M. Stuart and D. Dempster, *Verification Methodology Manual, for Code Coverage in HDL designs,*" Teamwork Int., Hampshire, U.K., 2000.

[5] I. Ghosh and M. Fujita, "Automatic test pattern generation for functional register-transfer level circuits using assignment decision diagrams," *IEEE. Trans. on Computer-Aided Design*, Vol. 20, No. 3, pp. 402-415, March 2001.

[6] B.T. Murray and J.P. Hayes, "Hierarchical test generation using precomputed tests for modules," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 594-603, June 1990.

[7] S. Ravi, G. Lakshminarayana, and N.K. Jha, "TAO: Regular expression based high-level testability analysis and optimization," in *Int. Test Conf.*, pp. 331-340, Oct. 1998.

[8] F. Ferrandi, G. Ferrara, D. Sciuto, and A.F. Fummmi "Functional test generation for behaviorally sequential models," in Proc. Design Automation and Test in Europe Conf., pp. 403-410, Mar. 2001.

[9] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, "Effective techniques for high level ATPG," in *Proc. Asian Test Symposium*, 2001.

[10] F. Fallah, S. Devadas, and K. Keutzer, "OCCOM: Efficient computation of observability-based code coverage metrics for functional verification," *IEEE. Trans. on Computer-Aided Design*, Vol. 20, No. 8, pp. 1003-1015, Aug. 1998.

[11] P.A. Thaker, V.D. Agrawal, and M.E. Zaghloul, "Validation vector grade (VVG): A new coverage metric for validation and test," in *Proc. VLSI Test Symp.*, pp 182-188, April 1999.

[12] P.A. Thaker, M.E. Zaghloul, and M.B. Amin, "Study of correlation aspects of RTL description and resulting structural implementations," in *Proc. Int. Conf. VLSI Design*, pp. 256-259, Jan 1999.

[13] ITC '99 benchmark suite, *http://www.cad.polito.it/tools/#bench, 1999*.

[14] T.M. Niermann and J.H. Patel. "HITEC: A test generation package for sequential circuits," in *Proc. European Design Automation Conf.*, pp. 214-218, Feb. 1991.

[15] M.S. Hsiao, E.M. Rudnick, and J.H. Patel, "Sequential circuit test generation using dynamic state traversal," in *Proc. European Design and Test Conf.*, pp. 22-28, Mar. 1997.

[16] F. Fallah, S. Devadas, and K. Keutzer, " Functional vector generation for HDL models using linear programming and 3-satisfiability," *IEEE. Trans. on Computer-Aided Design*, Vol. 20, No. 8, pp. 994-1002, Aug. 1998.

[17] M. C. Hansen, " Symbolic functional test generation with guaranteed low-level fault detection.," *Ph.D. Thesis, Dept. of EECS, Univ. of Michigan*, Ann Arbour, 1996.

[18] T.M. Niermann, W.T. Cheng, and J.H. Patel, "PROOFS: A fast, memory efficient sequential circuit fault simulator," in *Proc. Design Automation Conf.*, pp. 535-540, June 1990.

[19] H. Kim and J. P. Hayes, "Realization-independent ATPG for designs with unimplemented blocks," in *IEEE Trans. on Computer-Aided Design*, vol. 20, pp. 290-306, Feb. 2001.

[20] Verification Navigator, *http://www.transeda.com*.