Automatic Test Pattern Generation for Functional RTL Circuits Using Assignment Decision Diagrams

Indradeep Ghosh and Masahiro Fujita

Fujitsu Laboratories of America, Sunnyvale, CA 94086
{ighosh,fujita}@fla.fujitsu.com

Abstract

In this paper, we present an algorithm for generating test patterns automatically from functional register transfer level (RTL) circuits that target detection of stuck-at faults in the circuit at the logic level. To do this we utilize a data structure named assignment decision diagram which has been proposed previously in the field of high level synthesis. The advent of RTL synthesis tools have made functional RTL designs widely popular. This paper addresses the problem of test pattern generation directly at this level due to a number of advantages inherent at the RTL. Since the number of primitive elements at the RTL is usually lesser than the logic level, the problem size is reduced leading to a reduction in the test generation time over logic-level ATPG. A reduction in the number of backtracks can lead to improved fault coverage and reduced test application time over logic-level techniques. The test patterns thus generated can also be used to perform RTL-RTL and RTL-logic validation. The algorithm is very versatile and can tackle almost any type of single-clock design though performance varies according to the design style. It gracefully degrades to an inefficient logic-level ATPG algorithm if it is applied to a logic-level circuit. Experimental results demonstrate that over 1000 times reduction in test generation time can be achieved by this algorithm on certain types of RTL circuits without any compromise in fault coverage.

1. Introduction

The problem of automatic test pattern generation (ATPG) for sequential circuits has remained a difficult one in spite of great advances in ATPG techniques in recent years. The classical ATPG methods target the problem at the logic level and might require large amounts of computing time and resources to generate tests of even moderately sized sequential circuits [1]. By modeling circuits at a higher level, the number of primitive elements in the circuit is reduced, thus making the problem size more tractable. This allows larger circuits to be handled in less time.

In order to reduce the complexity of sequential ATPG various design for testability (DFT) schemes have been proposed that alter the circuit structure and functionality in the test mode to make the circuit easily testable. Two of the popular methods are scan design (full or partial) and built-in self-test (BIST) [1]. Though these techniques can increase the testability of a circuit considerably where sequential ATPG fails, they always come at a cost. There is an area overhead penalty, sometimes a performance penalty and power overhead penalty or a combination of all three. Moreover, test application time is increased very much in case of scan as scan vectors need to be sequentially fed into and scanned out of long scan chains.

In this paper, we try to tackle the sequential ATPG problem directly without any help of additional DFT hardware. Since the problem becomes too complicated at the logic level, we attempt to solve the problem at a higher level of abstraction where the number of primitive elements in the circuit is usually much lower thus reducing the problem size. Another motivation is the availability of RTL designs early in the design cycle. With the advent of robust, high quality RTL synthesis tools from a number of CAD vendors, designs are increasingly being done at the RTL. This can make the design process easier and cut design turn-around time drastically. Most of these descriptions describe the circuit in a cycle-accurate algorithm kind of fashion which is termed functional RTL. Our test generation algorithm directly acts on this kind of circuit descriptions. Since it can generate test vectors directly at the RTL, the test sets may also be used for doing RTL-RTL validation and RTL-logic validation.

The algorithm makes the following assumptions:
- The RTL design is in VHDL/Verilog and has a single clock line.
- The circuit does not have any complicated asynchronous behavior other than set/reset of flip-flops.
- The functionality of black boxes or intellectual properties (IPs) are stated such that each input of the block can be propagated to an output or a combination of outputs of the block in a fixed number of cycles and each output of the block can be justified from an input or combination of inputs of that block in a fixed number of cycles. If such a block is combinational and a test set for the block is provided, then the algorithm can attempt to test it. However, if the block is sequential in nature, then the algorithm cannot test it without some additional DFT hardware.
- Each finite state machine (FSM) description in the RTL circuit has a reset state or a single input line that takes the FSM to a fixed state when it is set or reset.

The algorithm first converts the HDL description into a graph-like structure called an *assignment decision diagram* (ADD). This data structure has been proposed previously to perform high-level synthesis [2]. The testing algorithm identifies arithmetic operation modules, logic arrays, registers, latches, memories, multiplexers, interconnect and random logic blocks from the ADD. Each of these elements is then tested by justifying test vectors to its inputs from the primary inputs (PIs) and propagating test responses from its outputs to the primary outputs (POs). This justification and propagation is done symbolically on the ADD representation with the help of a nine valued algebra and a branch and bound search procedure and requires backtracking similar to sequential ATPG. After this we have a set of justification and propagation paths from PIs to POs that exercises the elements deep inside the RTL circuit with test vectors applied at the PIs. This path which may span across many clock cycles is termed a *test environment* for the element under test. Finally, a test translation procedure uses test vectors from a well known test set for the RTL element or a precomputed test set from a test set library and plugs them into the *test environment* to obtain a system-level test set for the RTL element. The system-level test sets for various RTL elements are concatenated together to get the complete test set for the RTL circuit.

2. Previous Work

In the past, various attempts have been made to formulate ATPG algorithms that work on higher levels of design abstraction. A behavioral synthesis method has been proposed that generates testable RTL circuits from data/control flow intensive descriptions and also generates a system level test set as a byproduct [3]. At the RTL various testing techniques have been proposed. Most of them are integrated ATPG and DFT insertion techniques targeting BIST [4],[5],

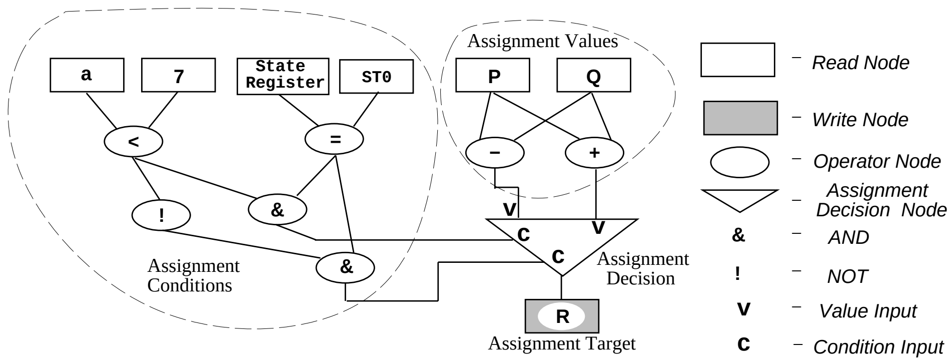

Figure 1: An Assignment Decision Diagram

scan design [6], or test multiplexers [7]. Also all the above techniques target structural RTL circuits and assume a clean separation between data path and control portions of the circuit. At the functional RTL, an interesting test generation approach uses a combination of integer linear programming and Boolean satisfiability to generate vectors targeting functional testing of RTL circuits [8]. This approach tries to find an exact solution to all justification and propagation problems and uses time-frame expansion to tackle sequential circuits. This can lead to an exponential increase in the complexity of the algorithm and the applicability of the algorithm to deep sequential circuits remains to be seen.

In this paper we have tried to address most of the above drawbacks by constructing a practical ATPG algorithm that can tackle most single-clock functional RTL designs. The algorithm gracefully degrades to an inefficient logic-level ATPG algorithm if given a functional RTL circuit that is pure Boolean in nature. This versatility is one of the main advantages of this algorithm over previous approaches. The algorithm can be fine tuned to generate greater fault coverage at the expense of CPU time much like logic-level sequential ATPG. Finally, we do report results on large, real-life industrial circuits unlike previous methods.

3. Assignment Decision Diagrams

Any digital system can be viewed as a set of computations on the PI values and contents of the internal data storage elements in the system. The results are in turn stored in internal storage elements or conveyed to the POs. Hence, a digital system can be represented as a set of conditional assignments to targets that represent storage elements or output ports in the form of an ADD, as shown in Figure 1. ADDs were first proposed for high level synthesis [2]. However, they can also be used to represent functional RTL circuits.

In Figure 1, the ADD representation consists of four parts: the assignment value, the assignment condition, the assignment decision, and the assignment target. There are just four types of nodes needed to represent an ADD: read nodes, operation nodes, write nodes and assignment decision nodes (ADN).

The *assignment value* part consists of read nodes and operation nodes. This part represents the computation of values that are to be assigned to a storage unit or output port. This value is computed from the current contents of the input ports or storage element or constants which are all represented by read nodes. The actual computation is represented as a data-flow graph which has the operation nodes that may be logical or arithmetic in nature. The *assignment condition* part consists of read nodes and operation nodes that are also connected as a data-flow graph. The end product of the computation is a Boolean value which is the guarding condition for the assignment value. The *assignment decision* part consists of an ADN. The ADN selects a value form a set of values that are provided at its value inputs. If one of the conditions to the ADN evaluates to *true*, then the corresponding input value is selected. The *assignment target* is represented by a write node. The write node is associated with the selected value from the corresponding ADN. A value will be assigned to the write node only if one of the condition inputs to the ADN evaluates to *true*. Since only one value can be assigned to a target at a time, all assignment conditions for a given target are mutually exclusive. This is shown in Figure 1.

In case of multi-state designs the corresponding ADD would contain a special storage unit called a *state register*. This state register represents the control step sequencer and has the same representation as any other storage unit. Assignments to the state register represent the sequencing of control steps, where each assignment value is a control step and each assignment condition represents the sequencing between steps.

The unique feature of an ADD is its capability to represent conditions and computations in a consistent and seamless data-flow fashion. Thus, operations in an ADD are ordered by their data dependencies only. With this capability an ADD can represent the most parallel representation of a given description. From the testing point of view this representation is suitable since it comes as close as possible to the final structural netlist without taking any synthesis decisions. It is a known fact that the structure of the final circuit is very important for generating vectors that detect stuck-at faults. Since data

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity TEST is
  port (  RST, CLK : IN std_logic;
          APORT, BPORT, CPORT, DPORT : IN std_logic_vector(7 downto 0);
          E : IN std_logic;
          OPORT : OUT std_logic_vector(7 downto 0) );
  end TEST;
architecture RTL of TEST is
  type STATE_TYPE is (S0, S1, S2, S3);
  signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
  signal A, B, C, D, F, G, O, NXT_A : std_logic_vector(7 downto 0);
  signal NXT_C, NXT_D, NXT_F, NXT_G : std_logic_vector(7 downto 0);
  signal NXT_B, NXT_O : std_logic_vector(7 downto 0);
  begin
    COMBIN: process(CURRENT_STATE)
      begin
        NXT_A <= A; NXT_B <= B; NXT_C <= C; NXT_D <= D;
        NXT_F <= F; NXT_G <= G; NXT_O <= O; OPORT <= O;
        case CURRENT_STATE is
          when S0 =>
            NXT_A <= APORT; NXT_B <= BPORT;
            NXT_C <= CPORT; NXT_D <= DPORT;
            NXT_G <= "00000000";
            NEXT_STATE <= S1;
          when S1 =>
            if ( C < D ) then
              NXT_F <= A + B; NEXT_STATE <= S2;
            else
              NXT_F <= A - B; NEXT_STATE <= S3;
            end if;
          when S2 =>
            NXT_G <= F + G;
            if ( E = '0' ) then
              NEXT_STATE <= S1;
            else
              NEXT_STATE <= S3;
            end if;
          when S3 =>
            NXT_O <= G - 3; NEXT_STATE <= S0;
        end case;
      end process;
    SYNCH: process(CLK, RST)
      begin
        if (CLK'event and CLK='1') then
          if (RST='1') then
            CURRENT_STATE <= S0;
          else
            CURRENT_STATE <= NEXT_STATE;
            A <= NXT_A; B <= NXT_B; C <= NXT_C; D <= NXT_D;
            G <= NXT_G; F <= NXT_F; O <= NXT_O;
          end if;
        end if;
    end process;
end RTL;
```

Figure 2: RTL VHDL description of circuit *Test*

and control are integrated, the testing of an element can be done in an integrated manner without caring about controller and data path separation. Also logic-level modules like multiplexers, adders, *etc* can be easily identified and tested from the description.

Note that for consistency, we are going to use VHDL as the high-level description language throughout this paper but similar arguments can be made about Verilog descriptions as well. In a VHDL description each process is converted to an ADD. In case of a combinational process and sequential process description of an FSM, the two processes are combined into a single ADD. The ADDs of processes are connected together by their input and output ports inside a component. There is also a separate block for the concurrent RTL statements. The components are in turn connected together to form larger components in a hierarchical fashion. The VHDL description of an FSM is shown in Figure 2 and the corresponding combined ADD is shown in Figure 3.

4. Motivational Example

In this section the RTL testing process is described with the help of the example circuit *Test* shown in Figure 2. First, an RTL ATPG algebra needs to be defined that is used to do symbolic justification and propagation on the RTL circuit. The algebra proposed here suitably modifies and extends similar concepts proposed earlier ([3], [7]). This makes the symbolic justification and propagation more versatile and powerful. Note that throughout this discussion the term variable has been used to signify an RTL variable, RTL signal or RTL net.

The ATPG algebra

The algebra consists of a set of nine symbols as follows:

- **Cg** (general controllability) of an RTL variable is the ability to control its value to any arbitrary value. *i.e.* if we are able to achieve *Cg* on a *n*-bit variable then it is possible to control this variable to any of 2^n values possible on that variable from the PIs.
- **C0** (controllability to 0) of a variable is the ability to control the variable to the value 0, *i.e.* “000..00” value in case of a multi-bit variable or 0 in case of a single bit variable.
- **C1** (controllability to 1) of a variable is the ability to control the variable to the value 1, *i.e.* “000..01” value in case of a multi-bit variable or 1 in case of a single bit variable.
- **Ca1** (controllability to all 1s) of a variable is the ability to control the variable to the value all 1s, *i.e.* “111...11” value in case of multi-bit a variable. In case of 2’s compliment arithmetic it also means controllability to the -1 value. *Ca1* of a single bit variable is equivalent to *C1*.
- **Cq** (controllability to a constant) of a variable is the ability to control the variable to any fixed constant value. *i.e.* in case of a *n*-bit variable we have to ability to control it to any one constant value out of the 2^n values possible on the variable. In case of a single-bit variable, it is the ability to control it to either 0 or 1
- **Cz** (controllability to the Z value) of a variable is the ability to control the variable to the high-impedance Z state. In case of a *n*-bit variable, it is the ability to control the variable to the “ZZZ...Z” vector.
- **Cs** (controllability to a state) of a state variable is the ability to control the variable to a particular state. Note that *Cs* is applicable to only state variables defined in an FSM.
- **O** (observability) of a variable is the ability to observe a fault at a variable. In case of a multi-bit variable, it signifies that some faulty value is present on the variable which is different from its correct value. In case of a single-bit value, it signifies the 1/0 fault *i.e.* the good value on the variable should be 1 and the faulty value 0 (it is the same as *D* in the *D-algorithm* used in logic-level ATPG [1]).
- **O’** (complement observability) is defined for single-bit variables only. It signifies the 0/1 fault *i.e.* the good value on the variable should be 0 and the faulty value 1 (it is the same as $\overline{D}$ in the *D-algorithm*).

Note that *Cg* on a variable subsumes *C0*, *C1*, *Ca1*, and *Cq* on that variable. Similarly either of *C0* or *C1* or *Ca1* subsumes *Cq*. We have chosen to represent only these few constants in the algebra because it is sufficient to control one of the input ports of any two-input arithmetic or logical operator supported in VHDL (synthesizable) to a suitable constant from this set in order to transfer test data from the other input port to the output port of the operator. Based on the above symbols we can formulate transformation rules for justification and propagation. For example *Cg* at the output of an addition operation can be transformed to *Cg* at its left input, *Cq* at its right input or *vice versa*. Similarly *O* at the input of a multiplication can be transformed to *O* at the output and either a *C1* or *Ca1* at the other input. Obviously *Cg* of all PIs are trivial and *O* or *O’* of all POs are trivial. Also, *Cq* for constant values are trivial.

Now, we have to introduce the notion of time frames. Also, to take care of bus splits and joins we have to introduce the bit indices of a variable. Thus we define an RTL *justification/propagation record* as a six valued set containing the ATPG objective, variable name, upper bit index and lower bit index of the variable, time frame value, and a state value which is present in case of *Cs* and empty otherwise. For example *Cg(a[7:0], 0, -)* implies that we need to meet the objective of general controllability on variable *a[7:0]* in time frame 0. Similarly *Cs(Curr_st, 3, S0)* means that we need to control the value of state variable *Curr_st* to *S0* in time frame 3 and so on. Note that bit-widths of state variables may be unknown at the functional RTL level.

State transition Graphs

In case of an FSM kind of description as in Figure 2, a state transition graph (STG) is also made along with the ADD only for the state variable. In the FSM the *reset states* (states which are reached by setting or resetting input lines to the FSM) are identified. Note that existence of such a state(s) is an assumption. Also identified are *input states*. These are the states where data is read from input variables to the process where the FSM exists. Similarly *output states* are states where the results of computations within the FSM description are written into variables which are outputs to the process in which the FSM

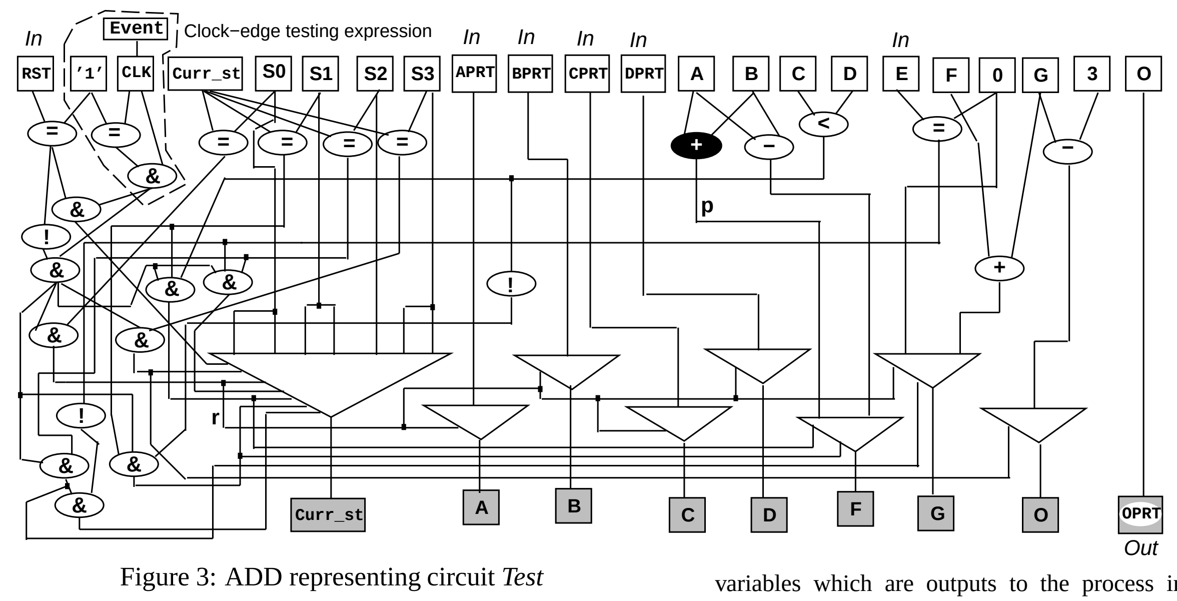

Figure 3: ADD representing circuit *Test*

exists. If there are multiple processes in a component, then these input and output variables can be found by doing a simple inter process connectivity analysis or a connectivity analysis between the process variables and the PI/PO ports. The state transition graph for the FSM in circuit *Test* along with the various type of states is shown in Figure 4.

Testing arithmetic modules

The first step for testing an RTL module is to find a set of justification (propagation) paths from (to) the module inputs (outputs) to the system PIs (POs). The VHDL description of the circuit is converted into an ADD representation. From the ADD the algorithm first identifies all multi-bit arithmetic operations in the circuit. It then tries to test each operation sequentially.

Consider the addition operation in state *S1* (highlighted in Figure 3). To test the operation, we need to control the two inputs of the operation from the primary inputs and observe the output response at a primary output. Also in the process we have to get the FSM to traverse from a reset state to an input state to the operation execution state *S1* and finally to an output state such that the justification and propagation objectives are met. The complete justification and propagation search is shown in Figure 5. The initial objectives are put in a stack. The *Cs* objective is required initially since the operation executes in state *S1* and this state is necessary for the data flow to happen through the operation. Objectives are transformed from the top of the stack according to the functionality of RTL modules. In case of multiple choices, selections are made randomly except in case of *Cs* objectives. In those cases the selection of the next or previous state is based on a greedy proximity heuristic that tries to guide the state transition sequence according to the required data flow as mentioned above. If the search fails in one direction, then it backtracks, abandons an intermediate decision and continues in a new direction. This is very similar to logic-level ATPG. There is a backtrack limit, and a time limit on each search.

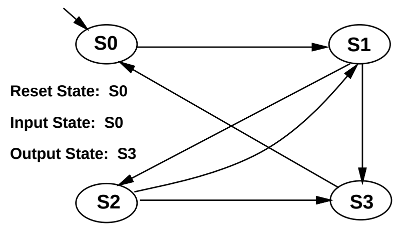

Figure 4: STG for circuit *Test*

As shown in Figure 5, first the state sequence is determined for justification, then the justification paths from the operation inputs to the PIs are determined followed by propagation paths which need to conform with the state transition graph. From the ADD viewpoint the following actions are taken during justification:

- If a read node is reached and it is not a PI or an input to the process, then the objective is shifted to the corresponding write node. In case of a PI the objective is satisfied. In case of a process input, the objective is shifted to a write node of another process (or concurrent RTL statements) which feeds the current process.
- If a write node is reached then the corresponding objective is shifted to one of the value inputs of the ADN feeding the write node and the C1 objective is required for the corresponding condition input. Over here there are multiple choices based on the number of value inputs in the ADN and there is a possibility of backtracking. If there is a clock-edge testing expression subtree (wait or event statements) in the expression tree that feeds the condition input then the time frame number is decremented by 1 (refer to Figure 3).
- If the output of an operation node is reached then it is transformed according to the functionality of the operation and the RTL algebra. The transformations for different RTL and logical operations are stored in a look-up table. Usually there are many choices and there is a potential for backtracks later.

Exactly reverse steps are taken in case of propagation. The only difference is while propagating from a control input of an ADN to its output the corresponding value input and complimentary condition value input need to have different values.

During the justification and propagation, a justification frontier is maintained in a stack with the objectives that needs to be justified. This is similar to the *J-frontier* used in the *D-algorithm*. Newly assigned objectives can conflict with objectives that have already been satisfied. For example if *Cg* is required on a variable in a time frame when there has already been a *Cg* assigned to it in the same time frame, then it leads to a conflict as it is impossible to control a variable to two arbitrary values in the same cycle. Another list of already satisfied objectives is also maintained. New objectives which are already satisfied are not entered into the stack. Similarly, during propagation a *O-frontier* is maintained. The time frame number starts at 0 on the initial objectives and slowly becomes more and more negative during justification and more and more positive during propagation.

Once the *test environment* is found for the addition operation, the most negative time frame number is assigned to 0 and the other time frame numbers are updated accordingly. Then a precomputed test set for the operation is used to plug in each test vector into the *test environment* and get a system-level test set that tests the addition operation. For example if (0, 5) is required in the (left, right) inputs of the addition operation as a test vector, we assign 0 to *A* and 5 to *B* and use the *test environment* to justify these values to the PIs, thus getting 0 at *APORT* and 5 at *BPORT*. This system-level test vector computation may require a series of arithmetic and logical transformations which is explicitly given by the *test environment*. PIs which are not used are fed with random values. The test sequence is as long as the largest time-frame number in the *test environment*. All test sequences obtained from all test vectors in the precomputed test set are concatenated together to get a system-level test set for the addition operation. All system-level test sets for all RTL elements are concatenated together to get the complete test set for the RTL circuit.

The precomputed test set for each type of arithmetic operation is stored in the library for each possible bit-width. This may look like a huge task but it is just a one time cost which needs to be incurred to test all RTL circuits. Since, at the functional RTL, it is impossible to know the structural implementation of the operation (it depends on the constraints placed during RTL synthesis), we tried to obtain test sets that do reasonably well for any implementation of the operation. We experimentally observed that using a test set for a most serial implementation of the operation in conjunction with a test set for the most parallel implementation, results in above 90% fault coverage for all types of implementation of the operation. For example for the addition operation the test set obtained by the combination of a ripple-carry adder test set with a carry-look-ahead adder test set results in good fault coverage for all types of adder implementation. As previously proposed it would be useful to get implementation independent test sets for operations if they are available [9].

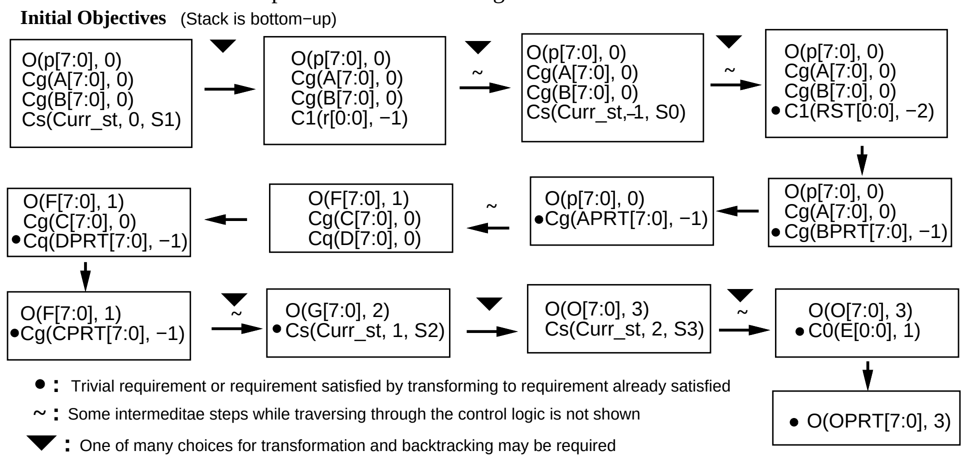

Figure 5: Justification/propagation for testing add operation

Testing logic arrays

After testing arithmetic operations the algorithm identifies multi-bit logic operations or logic arrays and tests them. Logic array test sets always fall within the constants defined in the RTL algebra. Thus each test vector can be individually justified and propagated without using the stringent *Cg* objective. For example, for an *OR* gate array with (left, right) input buses the test set is {(*Ca1*,*C0*), (*C0*, *Ca1*), (*C0*, *C0*)}. Once these objectives are set, the rest of the testing is the same as above.

Testing storage elements

Storage elements can be registers, latches and memories. From the ADD point of view storage elements are write nodes that have an ADN feeding it. If any of the condition inputs to the ADN has a clock-edge testing subtree then it can be implemented as a register, else it is a latch. To test all untested registers and latches, the pattern "000..00" (*C0*) and the pattern "111..11" (*Ca1*) is loaded into each of these elements and observed. For registers there are some additional two-pattern tests needed to test faults on the load line. However, these faults are detected when testing the ADN feeding the write node corresponding to this register in the ADD. This is discussed later.

Memories are usually formed by array declarations in the RTL. To test memories the *checker board* test is used. In this test a *test environment* is found out with *Cg* objective at the data and address variables of the memory and a *C1* objective on the write control port (*C0* if low active). This is used to write the checker board pattern into the memory. Then a *test environment* is found out with a *Cg* objective at address variable of the memory, a *C1* objective (*C0* if low active) on the read control port, and a *O* objective on the output port. This is used to read the test response from the memory. More complex memory tests can also be done in this manner.

Testing interconnect

The interconnect is in the form of either buses or multiplexers. In the ADD a two input multiplexer is represented as an ADN with two value inputs and complimentary condition inputs. Say the value inputs to the ADN are *v1* and *v2* and the condition input is *s* such that if *s* assumes the value 1, *v1* is chosen and *v2* otherwise. A multiplexer requires four patterns for complete testability. They are [*Ca1(v1)*, *C1(s)*], [*Ca1(v2)*, *C0(s)*], [*C0(v1)*, *Ca1(v2)*, *C1(s)*], and [*Ca1(v1)*, *C0(v2)*, *C0(s)*]. The algorithm tries to generate the above four *test environments* one after another. It tries to generate as many as possible if all the tests are not possible.

In case of a register, the register load is represented as a two input ADN on top of a write node representing the register variable as shown in Figure 6(a). In that case testing the ADN as above tests the register load line. In case of testing complex multi-input ADN nodes as shown in Figure 6(b), a *test environment* found out with *Ca1* requirement on each value input and *C1* requirement for the corresponding condition input. This type of ADN will be synthesized into a multi-input multiplexer, a multi-input multiplexer tree or a bus. Since it is impossible to figure out at this level of description, how the inputs will be ordered in the multiplexer tree, a select line stuck-at test can only be done if while asserting *C0* at a value input and *C1* at its corresponding condition input, all other value inputs are controlled to *Ca1*. This is usually too stringent

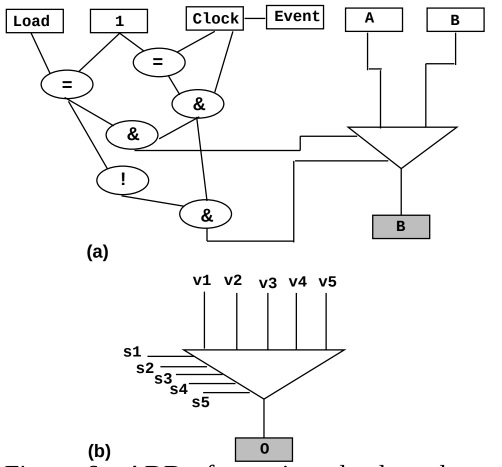

Figure 6: ADDs for register loads and complex ADDs

a requirement and the attempt to generate this *test environment* may fail. In that case in addition to the test of the form [*Ca1*(v_i), *C1*(s_i)] for each value input we also perform the test of the form [*C0*(v_i), *C1*(s_i)] for each value input and hope that the fault will be caught by random values at the other value input ports during fault simulation.

Testing random logic blocks

Random logic blocks are usually represented as logic controlling the condition ports of the ADNs. Also, if value inputs to an ADN are constants it will be synthesized as a random logic block. Random logic testing at the functional RTL level is quite difficult as there is no notion of structure. Hence, we try to cover the functionality of the circuit by doing observability enhanced statement coverage. This translates to identifying all random-logic ADNs, controlling each condition input signal of those ADNs to 1 and observing the write node at the output of the ADN.

Tackling bus splits and joins

There are various transformation rules that we use to tackle bus splits and joins based on the RTL ATPG algebra. For example during propagation, suppose bus *a[3:0]* splits into four individual bit lines. Then *O(a[3:0])* will be transformed to [*O(a[0:0])* & *O(a[1:1])* & *O(a[2:2])* & *O(a[3:3])*]. Similarly for justification if two bit lines join to form a bus, *C1(b[0:0])* and *C1(b[1:1])* will join to become *Ca1(b[1:0])*. Such transformations take place at *split* and *join* nodes which we identify from the ADD.

Note that while tackling single bit variables the combination of *C1*, *C0*, *O*, and *O'* makes the algorithm degenerate to a sequential version of a *D-like* algorithm. Thus ideally speaking the algorithm is versatile enough to tackle logic-level ATPG. However, in the absence of any modern-day heuristics its performance will be much worse than state-of-the-art logic-level ATPG tools.

Also note that the RTL ATPG process is not exact since it handles only a few constants for the sake of limiting the number of transformation choices across modules. For example all *O* requirements on bus lines are transformed to *O* requirements on bit lines. However, this may not be correct as all responses may not be of the 1/0 form. From the RTL algebra it is not possible to define the type of test response on individual bit lines in a bus. Only the information is stored that a fault response is present on the bus. Similarly while combining bit lines to form a bus during justification, if a constant is formed that is not represented by the RTL algebra, we simply put a *Cg* requirement on the bus. However, this can be an overkill as now all 2^n possible values need to be justified on the bus. In practice we observe that the algorithm works well even with its limited representation of the RTL constants. This is evident from the experimental results.

Tackling hierarchical designs

The algorithm can handle hierarchical designs. This is done with the help of the top level connectivity files that specify the connectivity among components. During justification when the justification frontier has reached the inputs of a component, the connectivity is used to transfer it to the output of another leaf component or to a PI. This might require some hierarchy traversal. However, the testing process occurs in all the leaf components and traversing hierarchy according to the connectivity is straight forward. Similar steps are taken during propagation where the aim is to finally reach a PO.

Tackling black boxes and IPs

Sometimes RTL circuits have components which are designated as black boxes. This can happen if a predesigned module is used whose logic-level representation is only present or while designing with IP cores whose RTL design is not made public. In such cases the justification and propagation will stop at the boundaries of these components unless some additional information is made available. We need two types of information for the RTL ATPG to succeed. First for each input of the black box a way should be provided to propagate its value to one or more outputs in a fixed number of cycles. The RTL ATPG will use this information to just propagate its *O-frontier* across the module. Similarly during justification for each output of the black box a way should be provided to justify its value from one input or combination of inputs in a fixed number of cycles. It is better if a number choices are available for justification and propagation which may be used in case of backtracks but at least one justification (propagation) path for each output (input) is

Table 1: Circuit size statistics for the example circuits

Circuit	*RTL*		*Logic level*	
	VHDL lines	Design type	#Gates	#Flip-flops
Paulin	130	flat	39558	227
Tseng	121	flat	22650	195
Dct	336	flat	13869	389
GCD	113	flat	1467	98
Barcode	156	flat	714	84
X25	122	flat	2250	227
Am2910	543	hierarchical	2109	140
GPIO	1002	hierarchical	1720	148
ALM	3504	hierarchical	8265	1490
EXE	8075	hierarchical	12327	939

a minimum.

If the black box is combinational in nature and a test bench is available, then the RTL ATPG algorithm can attempt to test the component like any other arithmetic operation. However, if the black box is a sequential design, then the clock for that component needs to be frozen for the cycles in which test data is being propagated from the PIs to the component inputs and component outputs to POs. This cannot be done without some additional DFT hardware.

5. Experimental Results

The RTL ATPG algorithm has been implemented implemented us-

Table 2: Test Generation Results

Circuit	*HITEC*			*STRATEGATE*			*RTL ATPG*		
	FC	TGen	TApp	FC	TGen	TApp	FC	TGen	TApp
	(%)	(sec)	(cycles)	(%)	(sec)	(cycles)	(%)	(sec)	(cycles)
Paulin	97.92	147002	752	99.71	101071	4499	99.72	138	4124
Tseng	98.43	52139	366	99.63	18481	2689	99.68	216	3429
Dct	90.01	74805	1696	89.83	106056	2528	96.50	739	3965
GCD	49.16	43964	258	85.73	192384	55823	94.31	498	4568
Barcode	63.41	9799	759	57.58	232336	24764	88.78	876	4080
X25	36.31	93592	151	57.27	131897	20817	85.35	1046	3561
Am2910	73.86	18723	1317	94.48	15125	4742	95.32	2765	3952
GPIO	99.41	57	1396	98.30	11078	5292	93.56	5543	690
ALM	22.53	58600	589	29.53	67854	1563	36.52	85654	1430
EXE	21.32	27300	992	44.12	399333	26722	40.83	585700	5689

ing the VHDL front end of the XE synthesis tool from YXI Inc. [10]. We have tested the RTL ATPG algorithm on ten example VHDL RTL circuits obtained from academia and industry. Out of these *Paulin*, *Tseng* and *Dct* are data-flow intensive filter type circuits extensively used in the literature [7], [5]. *GCD* (a circuit that calculates the GCD of two 32-bit integers), *Barcode* (a circuit used to scan barcodes from objects), and *X25* (a memory protocol handler) are control-flow intensive circuits. *Am2910* is a description of the microcontroller designed by AMD and is taken from [11]. *GPIO* (a general purpose input/output bus controller), *ALM* (a part of an ATM switch), and *EXE* (a memory controller) are industrial circuits..

The RTL designs were synthesized from VHDL by the Synopsys Design Compiler to gate-level netlists. Different scripts using different types of area, delay and map efforts were used to get various types of logic-level circuits with different types of optimization. This was done to ensure that the test vectors generated at the RTL are valid for different types of structural implementations of circuits. Table 1 shows the circuit statistics for the various examples used both at the RTL and logic level (one particular synthesis script was used for each circuit).

In Table 2 we compare our test generation results with two logic-level test generation tools. First we use the latest version of HITEC [12] which is a deterministic logic-level ATPG tool. HITEC has been run with a backtrack limit of 100,000. Then we use STRATEGATE a state-of-the-art genetic algorithm based logic-level test generation tool [13]. The fault coverage numbers for the RTL ATPG method are obtained by fault simulating the logic-level implementation of the circuits using PROOFS [12]. The CPU times are for an UltaSparc 60, 360 MHz machine with 512 MB memory. In the table *FC* means fault coverage, *TGen* is the test generation time and *Tapp* is the test application time or test sequence length in cycles. From the first line in the table it can be seen that over 1,000 times reduction in test generation time can be achieved by RTL ATPG over logic-level ATPG without any compromise in fault coverage. In fact the fault coverage for RTL ATPG is better than logic-level ATPG for 8 of the 10 circuits. However, the performance of the RTL ATPG algorithm degenerates as the circuit description at the RTL becomes more and more logic type.

Note that the test application time for RTL ATPG is on the higher side though a fair comparison cannot be made unless fault coverages for the two systems are comparable. Anyway, the reason behind this is that there is no RTL fault simulation. This results in many redundant vectors generated at the later stages of the algorithm.

We also did some experiments to confirm that the test vectors work well for different implementations of the same circuit by generating four or five versions of the same RTL circuit using different Design Compiler scripts and then obtaining fault coverage numbers on them with the same test set generated at the RTL. We observed that the fault coverage for the different versions varied by a maximum of 2%.

6. Conclusion

In this paper we have described a new and versatile RTL ATPG algorithm that can generate test vectors for almost any type of single-clock functional RTL design. The algorithm uses a data structure called assignment decision diagram that helps it to tackle control and data flow in an unified fashion and a nine-valued algebra that helps it to do justification and propagation at the RTL. The algorithm degenerates to an inefficient logic-level ATPG algorithm if it is fed a Boolean network. Currently the algorithm can provide a speed up of over 1,000 times over logic-level ATPG on certain types of designs. Moreover the patterns thus generated at the RTL can be used for RTL-RTL or RTL-logic validation. The performance of the algorithm degenerates as the circuit description becomes more and more logic type. Efforts are on to map effective logic-level ATPG heuristics into the algorithm so that the performance is comparable to logic-level ATPG even for logic type designs.

References

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

[2] V. Chaiyakul, D.D. Gajski, and L. Ramachandran, "High-level transformations for minimizing syntatic variances," in *Proc. Design Automation Conf.*, pp. 413-418, June 1993.

[3] S. Bhatia and N.K. Jha, "Integration of hierarchical test generation with behavioral synthesis of controller and data path circuits," *IEEE Trans. on VLSI Systems*, vol. 6, pp. 1789-1893, Dec. 1998.

[4] J.E. Carletta and C. Papachristou, "Testability analysis and insertion of RTL circuits based on pseudorandom BIST," *Proc. Int. Conf. Computer Design*, pp. 162-167, Nov. 1995.

[5] I. Ghosh, N.K. Jha, and S. Bhawmik, "A BIST scheme for RTL controller/data paths based on symbolic testability analysis," *Proc. Design Automation Conf.*, pp. 554-559, June 1998.

[6] S. Bhattacharya and S. Dey "H-Scan: A high level alternative to full-scan testing with reduced area and test application overheads," in *Proc. VLSI Test Symp.*, pp. 74-80, Apr. 1996.

[7] I. Ghosh, A. Raghunathan, and N.K. Jha, "A design for testability technique for RTL circuits using control/data flow extraction," *IEEE Trans. Computer-Aided Design*, vol. 17, pp. 706-723, Aug. 1998.

[8] F. Fallah, P. Ashar, and S. Devadas, "Simulation vector generation from HDL descriptions for observability-enhanced statement coverage," in *Proc. Design Automation Conf.*, pp. 666-671, June 1999.

[9] H. Kim and J.P. Hayes, "High-Coverage ATPG for datapath circuits with unimplemented blocks," in *Proc. Int. Test Conf.*, pp. 577-586, Oct. 1998.

[10] Y. Explorations, Inc. *Exploration Environment Tutorial*, Irvine, CA, 1999.

[11] S. Carlson, *Introduction to HDL-Based Design using VHDL*, Synopsys Inc., Mountain View, CA, 1990.

[12] 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.

[13] 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.*, Mar. 1997.