A Design for Testability Technique for RTL Circuits Using Control/Data Flow Extraction [*]

Indradeep Ghosh, Anand Raghunathan, and Niraj K. Jha
Department of Electrical Engineering
Princeton University, Princeton, NJ 08544

Abstract

In this paper, we present a technique for extracting functional (control/data flow) information from register transfer level (RTL) controller/data path circuits and illustrate its use in design for hierarchical testability of these circuits. This testing procedure and design for testability (DFT) technique is general enough to handle RTL control flow intensive circuits like protocol handlers as well as data flow intensive circuits like digital filters. It makes the combined controller-data path highly testable and does not require any external behavioral information. This scheme has the advantages of low area/delay/power overheads (average of 3.2%, 0.9% and 4.1%, respectively, for benchmarks), high fault coverage (over 99% for most cases), very low test generation times (because it is independent of bit-width), and the advantage of at-speed testing. Experiments show a 2-to-4 (1-to-3) orders of magnitude test generation time advantage over an efficient gate-level sequential test generator (combinational test generator that assumes full scan).

1 Introduction

Due to the ever-increasing complexity of integrated circuits, the problem of sequential test pattern generation has remained a difficult one in spite of good automatic test pattern generation (ATPG) techniques. The classical testing methods [1] target the problem at the gate level and might require huge amounts of computing time and resources to generate tests of modestly sized sequential circuits. By modeling circuits at a higher level, the number of primitive elements in the circuit is reduced, thus making the problem size more tractable.

In recent years, various behavioral and architectural schemes have been proposed to generate easily testable sequential circuits. These techniques may target BIST, scan or sequential ATPG [2]. However, while targeting a circuit for testing, the behavioral description is not available in many cases. Solving the problem at the gate level suffers from the problems mentioned above. Hence, in order to get the testability advantages of a higher level of abstraction, the RTL merits attention. Due to the popularity of macro-cell based designs, an RTL description is frequently available. Also, most of the above work has targeted data flow intensive designs and work on control flow intensive designs is limited. At the RTL, some schemes have also been proposed which target scan [3, 4] or non-scan DFT techniques [5]. Precomputed test sets have been used to test acyclic RTL circuits in [6]. In related previous work [7], RTL circuits obtained through behavioral synthesis have been targeted for hierarchical testability.

In this work, we propose a new methodology for making RTL circuits hierarchically testable using very little test hardware and without assuming they are obtained through behavioral synthesis. The only restriction imposed on the circuits is that they should have a separate data path and controller, and the controller should have a reset state. This scheme works by extracting a test control-data flow (TCDF) from the data path/controller circuitry and uses it to justify precomputed module test sets from the system inputs to module inputs and propagate error responses from module outputs to system outputs. When this is not possible, test multiplexers are added to the data path to increase its controllability and observability. A TCDF may not exactly correspond to the full control-data flow graph (CDFG) that the RTL circuit emulates. However, the test set derived with its help is also valid for the RTL circuit. The advantage of this scheme is that it is applicable to all RTL circuits conforming to the above assumptions whether it be data or control flow intensive. Based on experiments with benchmarks, the incurred area/delay/power overheads are very low. The fault coverage is very high for all the circuits. Due to the use of symbolic test generation in our scheme, the test generation time is independent of bit-width. This gives us several orders of magnitude advantage in ATPG time over efficient gate-level test generators. This scheme does not assume any scan at the controller-data path interface and the testing is actually done by using the data flow dictated by the controller. Finally, this scheme allows at-speed testing of the circuit.

2 Extracting TCDF information

In this section, we illustrate the extraction of functional information from RTL implementations and its application to test generation and DFT through several examples. We then discuss the DFT architecture.

2.1 A data flow intensive circuit

Figure 1 shows an RTL circuit obtained by synthesizing the CDFG of benchmark *Paulin* which has been popularly used in the literature. This particular RTL implementation was synthesized from a behavioral description using the HYPER [8] high-level synthesis system. However, as stated earlier, we will not be making use of the behavioral information available. The first step in the process is to extract the controller behavior from the controller circuit. This can be done by a state machine extraction program starting from the reset state. We used SIS [9] for this purpose. The controller part of the circuit is typically quite

[*]Acknowledgments: This work was supported by NSF under Grant No. MIP-9319269.

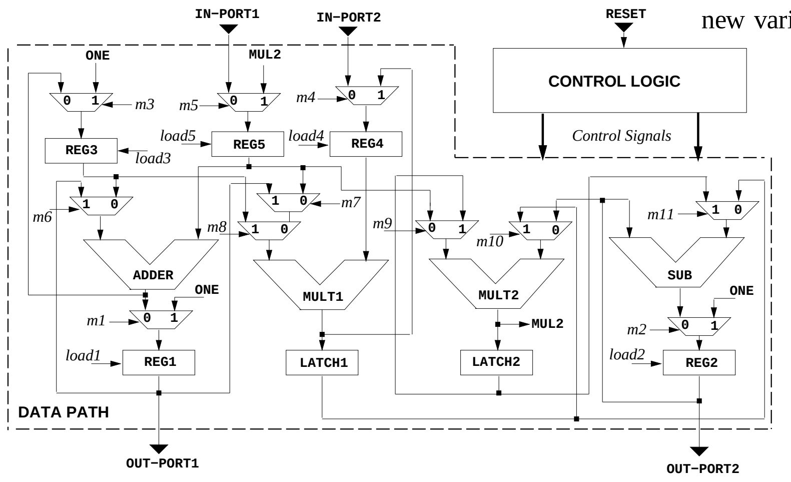


Figure 1: RTL circuit of Paulin.

small and this method is fast and efficient. Figure 2 shows the state table obtained from the controller circuit as well as the state transition graph. In the absence of conditionals, the state machine takes the form of a counter.

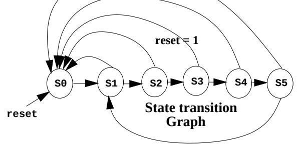
To obtain the TCDF, the basic idea is to extract operations executed in each cycle and keep track of variables that are present in each register or latch. We, in general, allow chaining, multicycling and structural pipelining. However, for simplicity, we omit discussions of these features here. We start with the input variables created in the first cycle. We identify all registers that load in the first cycle by analyzing the load signals in the state transition table. For our example, in cycle 1, all load signals are 1 and all registers load. Also, all latches load by default in all cycles. We next analyze the multiplexer tree that feeds each of these registers or latches and check if any input port is connected to the register/latch input in the first cycle. The multiplexer tree configuration in any cycle can be obtained by looking at their select signal values in the state table. We find that IN-PORT2 and IN-PORT1 are connected to REG4 and REG5, respectively, in cycle 1. Hence, two variables are born in these two registers. We call them *i1* and *i2*, respectively. A variable is live until its register loads again. Thus, some type of binding information is developed in each cycle and the variables bound to a particular register noted. We find that in cycle 1, REG1, REG2, REG3 are connected to hardwired constants. We name these *c1*, *c2* and *c3*, respectively, and their values are noted. The latches do not have any input ports or constants at their inputs. So they are ignored for now.

In cycle 2, we would like to identify the operations that take place. So we find the operand selected at each module's input ports by analyzing the multiplexer tree at its inputs. For example, in cycle 2, the left input of the adder is connected to REG1 and the right input to REG5. We check if both these registers have some live variables in them. Here, they do (*c1* and *i2*, respectively). We also check which registers (there may be many due to fanout), if any, at the module output load at the end of cycle 2, and if the input multiplexer configuration of these registers is such that this module output is selected. In case of the adder, REG1 indeed loads at the end of cycle 2. Moreover, its input multiplexer selects the adder's output. Hence, we create a new variable *r1* in REG1, and an operation in the TCDF, *c1 + i2 = r1*. We label this operation as *+1*. Similarly, we analyze all other modules to obtain the set of operations in cycle 2, as shown in Figure 3. The constant variables are shown with their actual values in brackets beside them. If there are no live variables in some register at a module input or if the module output is not loaded anywhere, then the operation is void and not added to the TCDF. For example, the subtracter does not have a valid operation associated with it in cycle 2, as the input latch associated with its right input (LATCH1) does not yet have a live variable. The binding information of operations to modules is also maintained. This procedure is repeated for each cycle until we reach a predefined *limit* on the number of states visited in the controller state machine as explained in Section 3.

There might exist operations in the generated TCDF which are not part of the original data flow. This is because latches will load by default in every cycle. So if there are live variables at a module input whose output goes to a latch, an operation will be created as well as a new variable in the latch. However, before this variable is ever used in the next cycle, it might be overwritten by another variable. For example, at the end of cycle 5, LATCH1 and LATCH2 load and this creates two multiplication operations in this cycle. However, these values are never utilized and the latches load new values at the beginning of the next iteration. Hence, these operations are spurious and the TCDF graph needs to be pruned. Sometimes, even a chain of spurious operations may be created. The pruning is performed by starting at the output variables in the TCDF and performing a backward traversal of the graph until all operations which are not in the support (*i.e.* transitive fanin) of these variables are deleted. The TCDF obtained for this circuit, after traversing five states, is shown in Figure 4. Symbolic justification and propagation of test vectors is performed using the TCDF in order to test the whole circuit hierarchically. Sometimes test multiplexers have to be added in order to facilitate this justification and propagation. This method is explained in detail in [7] and briefly described in Section 2.3.

2.2 A control flow intensive circuit

We next show how the procedure can be applied to a control flow intensive RTL circuit given in Figure 5. This circuit computes the *greatest common divisor* (GCD) of two numbers. The numbers are input at ports *XIN* and *YIN*, and the GCD is written to register *REGO* which is connected to the



State Transition Table

INPUT	*STATE*		*OUTPUTS*															
reset	PS	NS	load1	load2	load3	load4	load5	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10	m11
1	Any	S0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
0	S0	S1	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
0	S1	S2	0	0	0	1	0	0	0	0	1	0	0	0	1	1	1	0
0	S2	S3	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0	1
0	S3	S4	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	S4	S5	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
0	S5	S1	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0

Figure 2: The controller of Paulin.

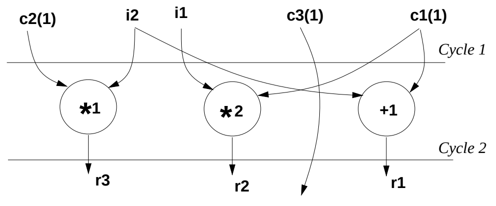
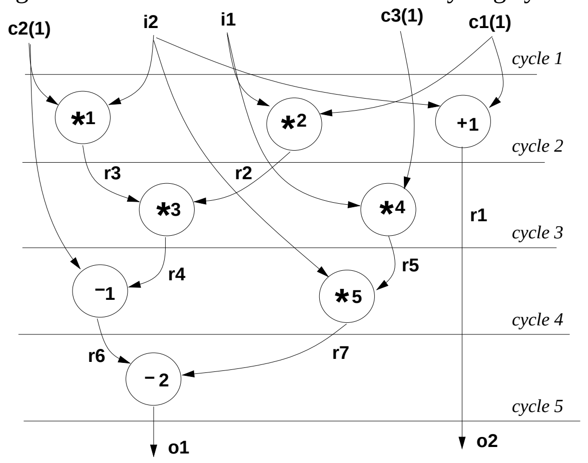


Figure 3: Partial TCDF obtained after analyzing cycle 2.



Binding information:

Inputs : i1, i2	*REG5 : i2*
Outputs : o1, o2	*LATCH1 : r2, r7*
REG1 : c1, r1, o2	*LATCH2 : r3, r4*
REG2 : c2, o1, r6	*ADDER : +1*
REG3 : c3	*MULT1 : *2, *4, *5*
REG4 : i1, r5	*MULT2 : *1, *3,*
	SUB : −1, −2

Figure 4: Final TCDF for Paulin.

output port. Since the number of cycles required to compute the GCD will vary according to the input values provided, the controller has an output signal *RDY* which specifies that the output is valid. There are three status signals going out from the data path to the controller, namely *C9, C10, C15*.

The problem in extending the approach of Section 2.1 to control flow intensive designs lies in the nature of the controller. A look at the state transition graph of the GCD controller given in Figure 6, shows that the state sequence depends heavily on the status signals generated. A fixed data flow consisting of a fixed number of cycles per iteration (as in the case of *Paulin*) is impossible to achieve here because the data flow depends on the inputs specified. For example, for one set of inputs, the subtracter may execute several operations before the output is ready. In some other case, the subtracter may not be exercised at all. One problem here is that the state machine is a Mealy machine. This means that in a particular state, the output signals might vary depending on the input status signals. Consequently, the set of operations executed in a particular state of this machine is not unique. This makes it more difficult to generate the TCDF information as outlined earlier. To solve this problem, we convert the Mealy machine into an equivalent Moore machine as shown in Figure 7 which can always be done (this conversion is only done for analysis purposes). In the Moore machine a state is associated with a fixed set of output values. Now we can proceed with the generation of the TCDF by analyzing this Moore machine.

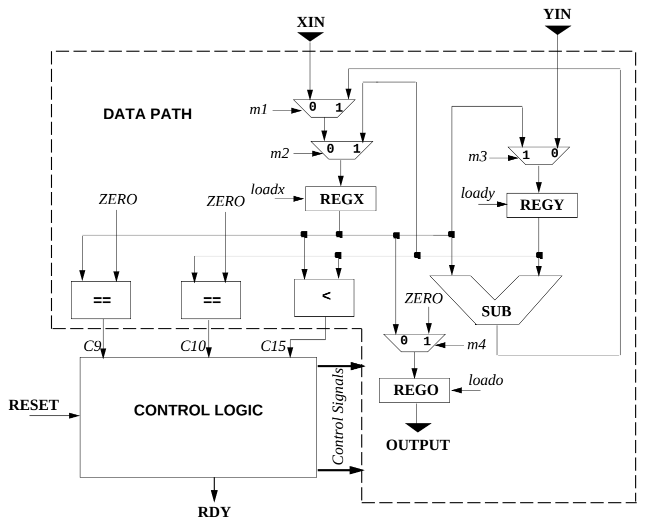
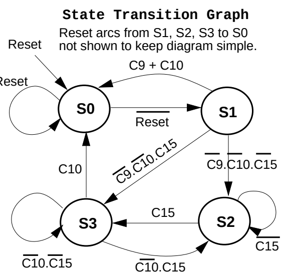


Figure 5: RTL circuit for GCD.

Note that the TCDF obtained from the Moore machine will also be valid for the Mealy machine, as the behavior of the two machines are equivalent. In the Moore machine the output of a particular row in the state table of Figure 6 is associated with the corresponding next state. Rows 1, (3,4) and 9 in the figure correspond to state $S0$, $S0'$, and $S0''$, respectively. The other states are as before.

As stated before, there is no fixed sequence of states in a particular iteration in this case. Hence, in order to test a module in the RTL circuit, we need to find out a possible state sequence which will provide us with a TCDF to test that module. The first step is to identify the *input states* where input registers get loaded from primary input ports. Here $S1$ is such a state. This is determined as before. Next, we find the *output states* where an output register is loaded with a value (not a constant). For example, output register REGO is loaded with a constant ZERO in state $S1$ and $S0'$, but gets loaded with a data path value only in state $S0''$. Loading a constant into an output register is useless from TCDF point of view. Therefore, only $S0''$ is an output state by our definition. Finally, we define *operation states* to be states where the module under test is exercised. This can be found by checking when the module output is getting


State Transition Graph
Reset arcs from S1, S2, S3 to S0 not shown to keep diagram simple.
Reset
Reset
C9 + C10
S0
S1
$\overline{\text{Reset}}$
C10
$\overline{\text{C9}}.\overline{\text{C10}}.\text{C15}$
$\overline{\text{C9}}.\overline{\text{C10}}.\overline{\text{C15}}$
S3
C15
S2
$\overline{\text{C15}}$
$\overline{\text{C10}}.\overline{\text{C15}}$
$\overline{\text{C10}}.\text{C15}$

State Table

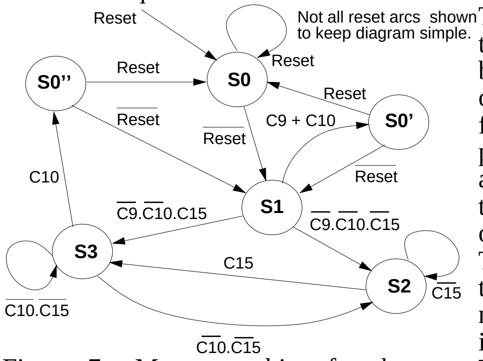
INPUTS				*STATE*		*OUTPUTS*							
Reset	C9	C10	C15	PS	NS	loadx	loady	loado	m1	m2	m3	m4	Rdy
1	x	x	x	ANY	S0	0	0	0	0	0	0	0	0
0	x	x	x	S0	S1	1	1	1	0	0	0	1	0
0	1	x	x	S1	S0	0	0	1	0	0	0	1	1
0	0	1	x	S1	S0	0	0	1	0	0	0	1	1
0	0	0	0	S1	S2	1	0	0	1	0	0	0	0
0	0	0	1	S1	S3	1	1	0	0	1	1	0	0
0	x	x	0	S2	S2	1	0	0	1	0	0	0	0
0	x	x	1	S2	S3	1	1	0	0	1	1	0	0
0	x	1	x	S3	S0	0	0	1	0	0	0	0	1
0	x	0	0	S3	S3	1	1	0	0	1	1	0	0
0	x	0	1	S3	S2	1	0	0	1	0	0	0	0

Figure 6: The Controller of GCD.

loaded into a register. For example, for the subtracter in GCD, the output is loaded into REGX in state $S2$ which is its operation state.

To get a TCDF for a module, we can obtain the sequence of states by going from an input state to an output state through an operation state, while traversing the state transition graph. For example, for the subtracter, we would like to go from $S1$ to $S0''$ through $S2$. We also need to figure out a path from the reset state to the input state, *i.e.* in this case from $S0$ to $S1$. If we want to keep the test application time as small as possible, we should immediately choose the shortest path between any two states and just concatenate them to get the complete path. For example, we find the shortest path between $S0$ and $S1$, then $S1$ and $S2$ finally $S2$ and $S0''$. To do this, any efficient shortest path algorithm may be used, *e.g.* Dijkstra's algorithm. One can argue that the sequence of states that the state machine will take will depend on the input values provided and it may not always be possible to traverse the shortest path. However, the test architecture that we present later gives us full controllability of the status signals during testing. Hence, we can actually dictate the path that the state machine will take during testing by controlling the external inputs.

Just finding the state sequence mentioned above may not be sufficient. There is an additional problem of data flow. For instance, in the above case, the shortest path is $S0 \rightarrow S1 \rightarrow S2 \rightarrow S3 \rightarrow S0''$. However, if we take the state machine through this path we obtain a TCDF which does not have a subtract operation. Hence, this is not a valid sequence from the point of view of testing the subtracter. In order to get a valid sequence, we need to backtrack along the path, do some loop unrolling, and explore another path which may not be the shortest path. For example, consider the sequence $S0 \rightarrow S1 \rightarrow S2 \rightarrow S3 \rightarrow S3 \rightarrow S0''$. This is obtained by backtracking one state from the previous path and unrolling the self-loop on $S3$ once. The TCDF that we obtain now is shown in Figure 8. The register-to-register transfers are shown by the assignment (:=) operator and the dashed lines and nodes are pruned from the TCDF as their outcome does not affect the final output. This TCDF is valid for testing the subtracter. During the TCDF generation, the status inputs that we need to dictate the flow is recorded, as shown in the figure. These inputs as well as the test vectors are fed into the test architecture from the primary input port, as explained later. In this manner, a TCDF is generated for each module in the circuit. For example, the TCDF obtained for testing the comparators is shown is Figure 9. During testing, the status outputs of the data path become observable at primary outputs because of the test architecture used. Hence, the comparison operations have outputs which become primary output



Reset
Not all reset arcs shown to keep diagram simple.
S0''
Reset
S0
Reset
Reset
$\overline{\text{Reset}}$
$\overline{\text{Reset}}$
C9 + C10
S0'
$\overline{\text{Reset}}$
C10
S1
$\overline{\text{C9}}.\overline{\text{C10}}.\text{C15}$
$\overline{\text{C9}}.\overline{\text{C10}}.\overline{\text{C15}}$
S3
C15
S2
$\overline{\text{C15}}$
$\overline{\text{C10}}.\overline{\text{C15}}$
$\overline{\text{C10}}.\overline{\text{C15}}$

Figure 7: Moore machine for the GCD controller.

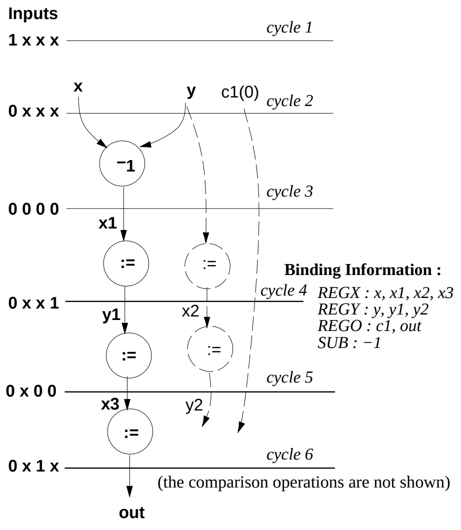


Figure 8: Successful TCDF for the subtracter.

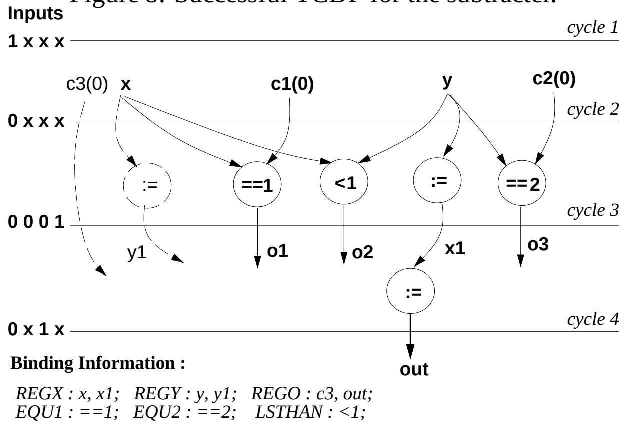


Figure 9: TCDF for testing the comparators.

variables. Test sequences are generated from each TCDF by using the method explained in [7]. A TCDF is not generated for testing registers or multiplexers separately. However, for each TCDF, the registers and multiplexers that are exercised in its path are targeted for testing.

Sometimes the RTL circuit might be such that no matter which sequence of states is taken, the required TCDF is never obtained. Consider the example of the barcode reader circuit - *Barcode*. A portion of the circuit is shown in Figure 10. Here the left input of the adder is never connected to a primary input. Hence, in order to get controllability over this input we add a test multiplexer as shown in the diagram. If the input to a module is a constant, as is the right input of the adder in this case, it is assumed

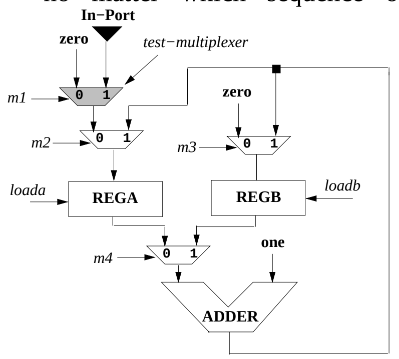


Figure 10: Part of RTL circuit *Barcode* showing the necessity of a test multiplexer.

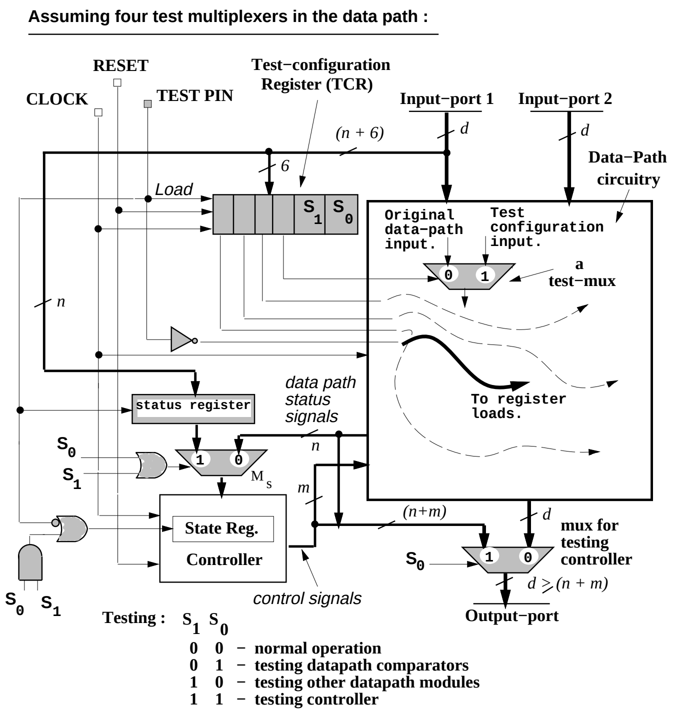


Figure 11: Test architecture used in the scheme.

to be swept away during logic synthesis. Hence, in this case, controlling one port and providing the test vectors corresponding to the swept adder is sufficient. Finally, we must point out that though we use a certain degree of backtracking here, the search never explodes. This is because each loop is allowed to be unrolled a limited number of times, and the number of states in the controller is typically not large. If we cannot find a valid TCDF, then we always have the option of adding a test multiplexer to create the required flow. During system-level test set creation, we also target the test multiplexers that we add for testability. The select input of the test multiplexer and one input (output) port is fully controllable (observable), as explained later. We just need a set of four vectors to test a multiplexer. Hence, the test multiplexers are inherently quite testable.

2.3 The test architecture

Various claims regarding the properties of the test architecture have been made earlier. We explain this architecture next. A low overhead solution is shown in Figure 11 where the added DFT hardware is shaded grey. (We assume one extra input pin that provides the *Test* mode signal.) The controller outputs are multiplexed with a data-path output-port to facilitate testing of the controller [7]. In case of circuits with conditionals, the status signals are made directly observable by multiplexing them with an output-port, and made directly controllable by feeding the status register input from an input-port. The DFT algorithm may have added test multiplexers to the data path in order to attain complete hierarchical testability. These multiplexers have to be configured while testing the data path components. For this, a *Test Configuration Register* (TCR) is added to the RTL circuit. The input of TCR is connected to the low-order bits of an input port. The *load* signal of TCR is connected to the *Test* pin. Its *reset* signal is connected to the controller *reset*. TCR feeds: (i) the *select* signals of the test multiplexers that are added to the circuit, (ii) two bits S_0 and S_1, that control the loading of the controller state register and selecting of the output multiplexer. The *load enables* of the data path and controller registers are qualified with the inverted *Test* signal to ensure that the data path and controller registers freeze their state while TCR and the status register are being loaded. Writing into TCR results in the circuit being reconfigured to provide controllability and/or observability as required. When we reset TCR, all the test-multiplexer select lines are 0. Hence, the normal data-path configuration exists. The *Test* signal and signals S_0 and S_1 should also be 0 for normal operation. While testing the controller, the *Test* signal, S_0 and S_1 are all set to 1. Then the controller inputs are controlled from primary inputs and its outputs observed at primary outputs. While testing the data path modules that generate the status signals, S_0 and S_1 are set to 1 and 0, respectively. Only at certain cycles when we want to reconfigure the test multiplexers or dictate the control flow do we assert the *Test* pin to load TCR and status register (default values should be fed to the register that we do not want to load). Similarly, while testing the other data path modules, we set S_0 and S_1 to 0 and 1, respectively. Since control chaining is assumed in the original design, the status register is shown as an overhead here. In designs where control chaining is not used, the status register is not an overhead.

Though the above discussion assumed edge-triggered flip-flops and a single-phase clock, similar arguments can be shown to hold for multi-phase and level-triggered clocking schemes as well. Also, in our method, unlike most previous high-level synthesis for testability methods, we do not assume any scan at the controller/data-path interface. Since all the tests can be fed at the normal clock speed of the circuit, at-speed testability is also made possible.

```
Procedure Test_RTL(RTL circuit $R$, limit)
{
  $Test\_Set$ = NULL;
  $L$ = extract_controller_FSM(R);
  $DP$ = data path of R;
  $L_m$ = Mealy_to_Moore(L);
  for (each module $M$ in $DP$) {
    $TCDF$ = Extract_TCDF($DP$, $L_m$, $M$, limit);
    if ($TCDF$ == NULL)
      $TCDF$ = Shortest_Path($L_m$, $M$);
    $DP'$ = DFHT($DP$, $TCDF$, $Test\_set$);
    $DP$ = $DP'$;    /* Update DP by adding
  }                                   test MUXs if any */
}
```

Figure 12: Procedure for testing the RTL circuit.

3 The DFT procedure

Figure 12 shows the top-level pseudocode that we use to generate the system-level test set of an RTL circuit. The procedure takes as input an RTL circuit R and an integer *limit* which specifies that while searching for a TCDF, the number of states in the search tree is not to exceed *limit*. How to find a suitable number for *limit* is discussed later on. We first extract the state transition information from the RTL circuit controller. If it is a circuit without condi-

```
Procedure Extract_TCDF($DP$, $L_m$, $M$, limit)
{
  $State\_stack$ = NULL;
  Push($S_r$, $State\_stack$); /* $S_r$ = controller reset state */
  $TCDF$ = NULL;
  $TCDF$ = Extract_TCDF_recurse($State\_stack$,
                                 1, TCDF, $L_m$, $M$, $DP$, $limit$);
  return $TCDF$;
}
```

Figure 13: Main procedure for extracting TCDF.

```
Procedure Extract_TCDF_recurse($State\_stack$, $depth$, $TCDF$, $L_m$, $M$, $DP$, $limit$)
{
  if ($depth$ > limit)
     return $NULL$; /* attempt failed */
  $S_{current}$ = get top of $State\_stack$;
  Update_TCDF($S_{current}$, $TCDF$);
  if (check_necessary_condition($TCDF$))
        return $TCDF$; /* attempt succeeded */
  else
   {
     for each neighboring state $S_{next}$ of $S_{current}$ in $L_m$ {
        Push($S_{next}$, $State\_stack$);
  if(Extract_TCDF_recurse($State\_stack$, $depth+1$, $TCDF$, $L_m$, $M$, $DP$)$\neq$NULL)
        {
        return TCDF ; /* attempt succeeded */
        }
     else /* attempt failed */
        {
        Undo_Update_TCDF($S_{next}$, $TCDF$);
        Pop($State\_stack$);
        }
     }
        return NULL;
   }
}
```

Figure 14: Recursive function for extracting TCDF.

tionals, the state machine is already in the form of a Moore machine. Else, we convert the state machine to an equivalent Moore machine. Well-known algorithms exist for this purpose. Next, for each module in the data path (*DP*) of the RTL circuit, we try to generate a TCDF which contains an operation mapped to that module from the Moore machine L_m and *DP*. Procedure *Extract_TCDF*, which is explained later, is used to find such a TCDF. It is possible that even after a long search with the help of the state machine, such a valid TCDF is not found as shown in the *Barcode* example (Fig. 10). In such cases we use the shortest path algorithm to get the shortest path from an *input state* to an *output state* through some *operation state*. We extract the TCDF generated while traversing this shortest path. In this TCDF, operations are not pruned as otherwise the operation under test will not appear at all.

After TCDF extraction, it is used in the *design_for_hierarchical_testability* (DFHT) procedure to add test multiplexers to the RTL circuit in such a way that the TCDF generated in the modified RTL circuit while traversing the specified path has the necessary hierarchical controllability and observability for the operation under test (hence, the module to which it is mapped). The DFHT procedure takes as input an RTL circuit and a TCDF. It identifies variables mapped to registers in the RTL circuit which are bad from hierarchical controllability or observability point of view by doing a hierarchical testability analysis on the TCDF. This analysis is done symbolically. Hierarchical controllability and observability of a variable are Boolean parameters, *i.e.* they only take the values 0 and 1. When hierarchical controllability (observability) is 1, it implies that it is possible to control the variable to any arbitrary value from system inputs (observe any error response on it at system outputs). The hierarchical controllability (observability) of a “bad” register can be made 1 by multiplexing it with a hierarchically controllable (observable) variable using a test multiplexer. This procedure is explained in detail in [7].

In the case of a circuit without conditionals, it is possible to construct the whole data flow graph and test all modules in the circuit using it, as in example *Paulin*. For such cases, we generate a single global TCDF by traversing the whole state machine and apply DFHT to it. The system-level test set is a byproduct of the DFHT procedure and for each module the test set generated is appended to the global test set. The test multiplexer added to the RTL circuit gives it some additional data flow. This information may be useful while testing other modules because one test multiplexer may solve the hierarchical controllability or observability problems of many variables simultaneously. Hence, the data path is updated in each iteration after a test multiplexer is added.

The *Extract_TCDF* procedure shown in Figure 13 takes as input L_m, *DP*, $limit$ and the module under test. From the procedure a recursive search procedure (*Extract_TCDF_recurse*) is called whose pseudocode is given in Figure 14. In the recursion, a stack of states is maintained. States are pushed onto the stack in the order they are visited. The search starts from the reset state. The search continues until a valid TCDF for module *M* is found or if all possible paths of length $limit$ have been exhausted without finding a $TCDF$. This is a type of depth-first search where the same state here might be visited multiple times until the search stops. This helps us to unroll loops in the Moore machine which might be necessary, as shown in the *GCD* example. In each step of the search, a partial TCDF is maintained, as described in Section 2.2, and augmented in each step with the help of procedure *Update_TCDF*. When we reach an output state, we see if any operation mapped to module *M* is in the support of that output. This is done in procedure $check\_necessary\_condition$. If such an operation exists in the TCDF, then we have found a successful TCDF for testing *M*. Otherwise we backtrack and explore other paths. In such cases, the procedure *Undo_Update_TCDF* is used to undo the changes made earlier during the search.

The above search procedure is exhaustive and its worst-case complexity is exponential in the number $limit$. However, the number of states in a state machine controller is typically small. Hence, the search space in practice is generally quite small. The $limit$ that was used in experimental

Table 1: Circuit size and DFT hardware statistics.

Circuit.	Bit-width	# lits	# flip-flops	# test MUXs	CPU time (sec.)
Paulin	32	24562	260	1	3.5
Elliptic	16	22265	229	6	93.9
Tseng	32	15053	197	2	27.5
Chemical	16	20924	246	5	236.5
Dct_lee	8	9354	111	2	156.3
Pr1	8	11203	100	4	720.1
GCD	16	1267	51	0	0.0
Barcode	16	1881	118	3	1.3
X25	16	2421	116	2	2.2

results was twice the number of states in the Moore machine. This allowed for at least one unrolling of any loop and was sufficient for all benchmarks. If TCDF generation fails with this limit, then it may be better to add test multiplexers to the data path rather than waste test generation and application time by increasing *limit*. Also, we developed a few heuristics to guide the search for a particular module. We calculated the length of the shortest path from a state to an *operation state*. Let this be l_{op}. We also calculated the shortest path length from an *operation state* to any *output state*. Let this be l_{ou}. The l_{op} and l_{ou} values can be found by an all-pairs-of-shortest-path algorithm. When developing the TCDF, our aim was to guide the search from a reset state to an *operation state* and then from the *operation state* to an *output state*. Hence, during the recursive procedure, until we reach an *operation state*, the next state among a lot of possible states is chosen as the one which has the least l_{op} value. Once an *operation state* is reached the next state is chosen as the one which has the least l_{ou} value.

4 Experimental results

Table 2: DFT hardware placement overheads.

Circuit.	*Area*			*Delay*			*Power*		
	Orig.	*Mod.*	*Ovhd.* (%)	*Orig.* (ns)	*Mod.* (ns)	*Ovhd.* (%)	*Orig.* (mW)	*Mod.* (mW)	*Ovhd.* (%)
Paulin	294332	296312	0.7	231.4	231.8	0.2	137.2	138.0	0.6
Elliptic	260554	274213	5.2	177.4	178.8	0.8	114.5	120.3	5.1
Tseng	176932	178321	0.8	23.8	23.8	0.0	71.3	75.8	6.3
Chemical	241974	252644	4.4	41.6	42.1	1.2	95.2	100.3	5.4
Dct_lee	98176	101736	3.6	23.1	23.1	0.0	43.4	45.7	5.3
Pr1	118272	123251	4.2	28.3	28.3	0.0	14.2	14.8	4.2
GCD	15520	16436	5.9	55.4	57.0	2.9	2.9	3.0	3.1
Barcode	30368	32497	7.1	33.6	35.1	4.5	8.2	8.7	6.1
X25	33952	35950	5.9	87.2	88.2	1.2	9.4	9.9	5.6

In this section we present experimental results based on the application of our DFT procedure to nine example RTL circuits. Among these, *Paulin* is as shown before. *Elliptic* is a fifth-order elliptic wave filter. *Tseng* is a well-known example from the literature. *Chemical* is a type of IIR filter used in a chemical plant controller. *Dct_lee* performs discrete cosine transform. *Pr1* implements a rotation-based discrete cosine transform. All these examples are data flow intensive. Of these examples, *Paulin* and *Elliptic* are area-optimized circuits synthesized by HYPER [8]. *Tseng* and *Chemical* are synthesized by SCALP(delay) [10] and are optimized for delay. *Dct_lee* and *Pr1* are power-optimized circuits synthesized by SCALP(power). The remaining three examples are control flow intensive. *GCD* and *Barcode* are as discussed before. *X25* is a memory protocol handler. These circuits were obtained from the industry. The area, delay and power results were obtained after technology mapping the gate-level implementation of the RTL circuits using the *stdcell2_2.genlib* cell library in SIS[9] logic synthesis system.

Table 1 shows the characteristics and specifications of these circuits. The DFT overheads are reported in Table 2 and the test generation results are shown in Table 3. In Table 1, in Columns 2, 3 and 4, the bit-width, the literal-counts of the original technology-mapped circuit and the number of flip-flops are given, respectively. The number of test multiplexers added to the circuit by the DFT procedure is given in Column 5. This number does not include the multiplexers in the test architecture which are added by default to all circuits, and which have been taken into account while calculating the overheads in Table 2. In Column 6, the CPU time required to extract the TCDF and place the DFT hardware is given. All CPU times are measured on a SPARCstation 20 with 128 MB memory. In Column 2 of Table 2 the original area of the circuits after technology mapping is given. This is a relative figure obtained from the layouts of the standard cells used and hence has no units. Column 3 shows the area after the circuits have been modified by the test architecture and hardware. The percentage overhead is given in Column 4. In Columns 5, 6 and 7, the corresponding figures for delay are provided. The delay represents the clock period in *nanoseconds*. The delay overheads are small, even for delay-optimized circuits (*Tseng* and *Chemical*), because the test multiplexers can frequently be placed off the critical path. The corresponding power figures are given in Columns 8, 9 and 10. The power consumption is measured in milliwatts and is determined through SIS at the logic level using the mapped capacitance and zero delay model. More accurate power estimation could not be done due to the size of some of the circuits used. The average area, delay and power overheads are only 3.2%, 0.9%, and 4.1%, respectively (the average is calculated based on total area/delay/power of all the examples for the original and modified cases).

In Table 3, the testability results for the circuits augmented by our DFT method is shown. We compare our method of hierarchical testing against HITEC [11], an efficient gate-level sequential test pattern generator. Columns 2 and 3 show the fault coverage and test generation times obtained by running HITEC on the original circuits not modified by our DFT hardware. Columns 4 and 5 give the fault coverage and test generation times obtained by running HITEC on the circuits modified by our DFT scheme. Columns 6 and 7 show the corresponding numbers obtained by our method of hierarchical testing of the modified circuits. The fault coverage for our scheme was obtained by fault simulating the gate-level implementation of the controller/data path with our system-level test set using PROOFS [11]. The fault coverage obtained by our method is higher in all the cases and roughly 99% or above for nearly all the examples. Whereas our method was used to generate the data path test set, HITEC was used to derive the controller test set from its gate-level description. Then the two test sets were concatenated to derive our system-level test set. This is valid because the inputs (outputs) of the controller are directly controllable (observable) through the extra multiplexers added for this purpose. As the controller is a small fraction of the total circuit, the ATPG on the controller alone takes a very small amount of time. However, the test generation time reported for our scheme includes the test generation times for both the controller and data path. It is 2-to-4 orders of magnitude smaller than HITEC. By comparing the experimental results on the original and modified circuits it is clear that even gate-level sequential test pattern generation benefits from our scheme in terms of both fault coverage and test generation times.

Table 3: Testability results.

Circuit.	*HITEC*				*Our Method*	
	Orig. Ckt.		*Mod. Ckt.*		*Mod. Ckt.*	
	Fault Cov. (%)	*Test Gen.* (sec)	*Fault Cov.* (%)	*Test Gen.* (sec)	*Fault Cov.* (%)	*Test Gen.* (sec)
Paulin	89.62	66412	95.92	66432	99.70	12.1
Elliptic	57.56	80304	98.96	60517	99.61	85.6
Tseng	78.48	59646	97.07	51347	99.02	26.4
Chemical	54.31	76543	97.53	65876	99.16	56.3
Dct_lee	71.32	61392	99.35	40525	99.45	72.3
Pr1	90.62	60188	99.39	40576	99.41	56.2
GCD	86.42	6752	98.81	32	98.89	0.3
Barcode	31.01	56813	98.78	1402	98.88	6.2
X25	50.70	18762	93.96	6716	98.72	5.9

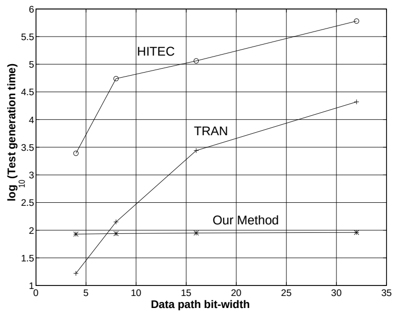


Figure 15: Log-linear plot of test gen. time *vs* bit-width

We also performed some experiments in which we investigated how the test generation time for our scheme would compare against the full scan method which just requires gate-level combinational test generation. We used an efficient combinational test generator, TRAN [12], for this purpose. Since the combinational test generator assumes full scan, such testable circuits would incur significant additional overheads as compared to our method. We found that our test generation times were 1-to-3 orders of magnitude smaller than those for TRAN. In order to examine how different methods scale with the data path bit-width, we performed experiments for various controller/data path examples and different bit-widths. Figure 15 shows log-linear plots of the test generation time for the controller-data path by our method, HITEC and TRAN *versus* the bit-width of the data path for the *Elliptic* example synthesized by HYPER and augmented by our method. The *x*-axis is the bit-width, and the *y*-axis is log_{10}(CPU seconds taken for test generation). These plots indicate that the CPU time required for test generation by HITEC and TRAN increases drastically with bit-width. However, the test generation time for our method remains practically constant with increasing bit-width. At a bit-width of 32, our scheme can be seen to have a two (four) orders of magnitude advantage in test generation time over TRAN (HITEC).

5 Conclusions

In this paper we introduced an effective and practical DFT scheme that can be applied to data path-controller type of RTL circuits. The key feature of this technique is that it analyzes the data path and controller of an RTL circuit and extracts a test data and control flow graph. This graph is then used to test the circuit hierarchically by justifying and propagating precomputed test sets of modules in the circuit from system inputs and propagating the output responses to system outputs. If it is not possible to do so then test multiplexers are added at suitable places to increase the hierarchical controllability and observability of the circuit during the test mode. To ease the task of TCDF generation we have defined several techniques to be applied to the controller state machine and data path and defined a test architecture which is necessary for the process. The advantages of this technique are: (i) low area, delay and power overheads, (ii) high fault coverage, (iii) several orders of magnitude of reduction in test generation time over gate-level ATPG, and (iv) at-speed testability.

References

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

[2] K.D. Wagner and S. Dey, “High-level synthesis for testability: A survey and perspective,” in *Proc. Design Automation Conf.*, pp. 131-136, June 1996.

[3] J. Steensma, F. Catthoor, and H. De Man, “Partial scan at the register transfer level,” in *Proc. Int. Test Conf.*, pp. 488-497, Sept. 1993.

[4] 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.

[5] S. Dey and M. Potkonjak, “Non-scan design-for-testability of RT-level data paths,” in *Proc. Int. Conf. Computer-Aided Design*, pp. 640-645, Nov. 1994.

[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] I. Ghosh, A. Raghunathan, and N.K. Jha, “Design for hierarchical testability of RTL circuits obtained by behavioral synthesis,” in *Proc. Int. Conf. Computer Design*, pp. 173-179, Oct. 1995.

[8] J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, “Fast prototyping of data path intensive architectures,” *IEEE Design and Test.*, vol. 8, pp. 40-51, June 1992.

[9] E. Sentovich, K. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli, “Sequential circuit design using synthesis and optimization,” in *Proc. Int. Conf. Computer Design*, pp. 328-333, Oct. 1992.

[10] A. Raghunathan and N.K. Jha, “An iterative improvement algorithm for low power data path synthesis,” in *Proc. Int. Conf. Computer-Aided Design.*, pp. 597-602, Nov. 1995.

[11] 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.

[12] S.T. Chakradhar, V.D. Agrawal, and S.G. Rothweiler, “A transitive closure algorithm for test generation,” *IEEE Trans. Computer-Aided Design*, vol. 17, pp. 1015-1028, July 1993.