Automated Modeling of Custom Digital Circuits for Test

Soumitra Bose
Intel Corporation, Folsom, CA 95630
Design Technology (Logic & Test)
email: bose@ieee.org

Abstract

Models meant for logic verification and simulation are often used for ATPG. For custom digital circuits, these models contain many tristate devices, which leads to lower fault coverage. Unlike other research in the literature, the modeling algorithms presented in this paper analyze each channel connected component[1] in the context of its environment, thereby capturing the relationship among its input signals. This reduces the number of tristates and increases the modeling efficiency, as measured by fault coverage. Experimental results demonstrate the superiority of this approach.

1. Introduction

Purely combinational circuits are far more testable than those that have tristates. None of the algorithms in the literature can generate combinational models automatically for the class of custom circuits that are often encountered in practice. Tristate models typically exhibit much lower fault coverage due to propagation of unknown (X) values. We present new algorithms for extracting models that have minimal tristate devices. These algorithms perform well on custom digital circuits that have pass transistors, multiplexers, bus drivers, and custom latches, as demonstrated by experimental results.

Some of the salient features of the algorithm presented in this paper include:

1. Models are not generated in isolation and depend on the surrounding environment of a Channel Connected Component (CCC). This is illustrated with examples in Section 2.

2. The complexity of Boolean functions, as measured by the size of Binary Decision Diagrams (BDD) [5], is constrained due to incremental depth-based analysis. Since the complete logic cone at a signal is not known (because latches are not known apriori), building entire BDDs is not an option.

[1] A Channel Connected Component is a group is transistors that are connected by their source and drain terminals, unless these terminals happen to be a primary input or power or ground node

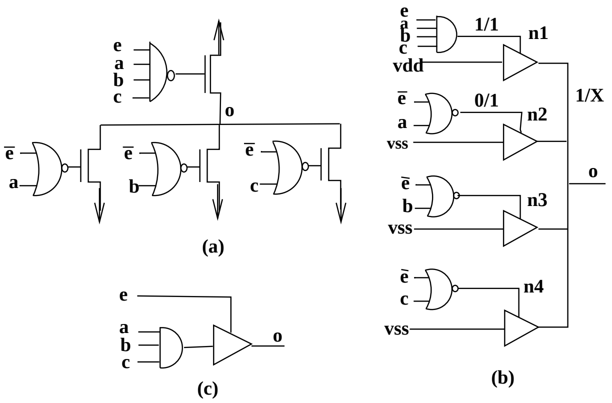

Figure 1: Incorrect modeling of bus driver

3. The automated algorithms for extraction of bus drivers, multiplexers and latches are completely new.

To demonstrate the effectiveness of these algorithms, we compare a specific modeling feature with the learning capability of a commercially available ATPG tool. The impact of these algorithms is evident from the results presented in Section 6.

2. Motivation

To illustrate the significance of modeling, we consider the tristatable bus driver shown in Figure 1(a). Signal e is an enable signal and a, b and c are data signals that are driven onto the bus node o. A tristate model for the circuit is shown in Figure 1(b). This model would be generated if the four transistors in the CCC are analyzed in isolation, and would be the case for other known modeling algorithms [6, 7, 9, 11]. The drawback with the model is evident by considering the $n2@1$ fault. In order to activate this fault, either one or both of a and $\overline{e}$ must be 1. In order to propagate this fault effect, outnode node o must be 1 in the good circuit. This implies all of e, a, b and c must also be 1 in the good circuit. But this would be true in the faulty circuit also, thereby driving node o to X. Hence, the s@1 fault at node $n2$ is untestable in this

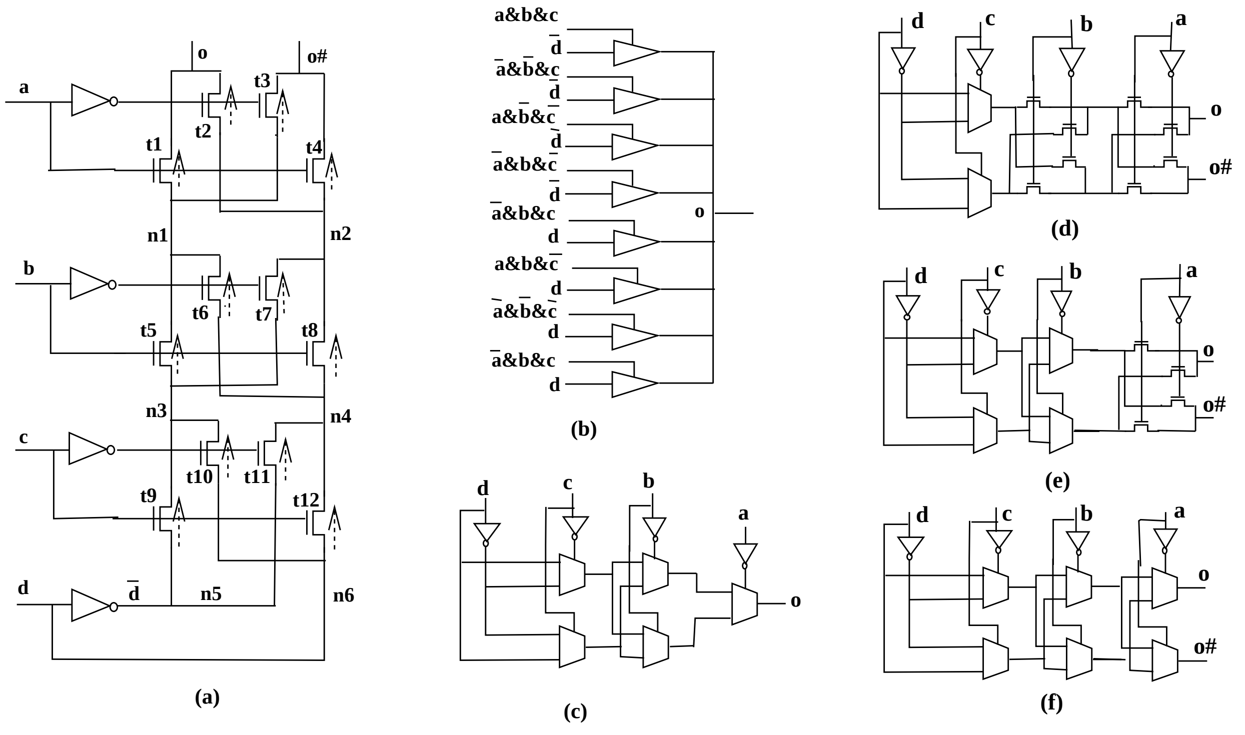

Figure 2: Parity circuit example

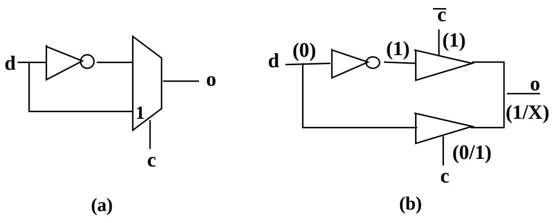

Figure 3: Incorrect modeling of simple multiplexer

model. The model shown in Figure 1(c) is superior and can be derived only if the logical relationship among the input signals to the CCC are analyzed.

Another example of improved modeling is shown in Figure 2(a). For output signal *o*, a simple model is shown in Figure 2(b). An improved model for this signal is shown in Figure 2(c). The latter model has a much higher fault coverage than the former. To understand the basic reason for the lower fault coverage, we consider the single stage multiplexer shown in Figure 3(a). For the tristate model shown in Figure 3(b), fault *c*@1 is untestable. The model shown in Figure 3(a) is fully testable. There are many faults of similar nature in the circuit of Figure 2(a).

3. Previous Work

Wadsack's 1978 paper [16] gives a logic model for MOS bus. The memory state is modeled using a latch whose control (or clock) input is derived from the bus control signals using a combinational network. Jain and Agrawal [12] used a modeling element, called *B-block*, which can be customized to model either a 0-dominant or a 1-dominant bus as well as provide a high-impedance state when all drivers are turned off. Although attempts have been made to represent the bidirectional signal flow [1], these models are most popular when the direction of signal-flow is presumed to be fixed.

Switch-level models [2, 4, 6, 7] provide accurate modeling of bidirectional behavior within channel-connected transistor blocks. Highly successful for fault-free simulation, this model has been used for test generation [10, 13]. At least two fault simulators for circuits modeled at the transistor-level have also been written: FMOSSIM [15] for simulating stuck faults, and a more recently reported one for delay faults [3].

4. Logic and tristate models

This section outlines the various algorithms used for model extraction, and will be described in various stages. Combinational gates are first recognized, and the result of that analysis is used in subsequent stages for the analysis of tristatable devices, multiplexers, and latches.

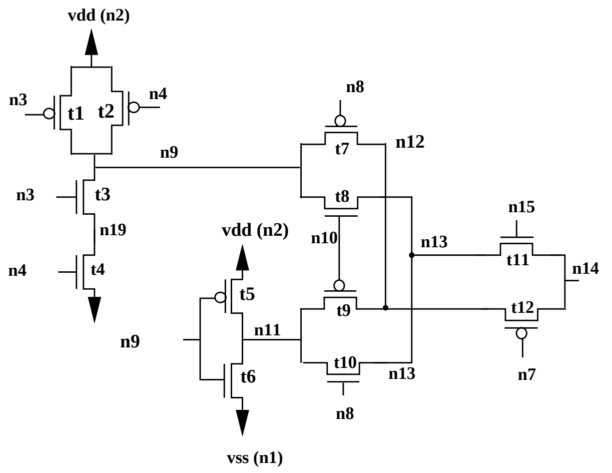

Figure 4: Example of a CCC

Table 1: AND-OR reduction for Figure 4

before reduction			after reduction		
switch	N1	N2	switch	N1	N2
s1=t1	n2	n9	s13=(t1+t2)	n9	n2
s2=t2	n2	n9	s14=(t3.t4)	n9	n1
s3=t3	n9	n19	s5=t5	n11	n2
s4=t4	n1	n19	s6=t6	n11	n1
s5=t5	n2	n11	s7=t7	n9	n12
s6=t6	n1	n11	s8=t8	n9	n13
s7=t7	n9	n12	s9=t9	n11	n12
s8=t8	n9	n13	s10=t10	n11	n13
s9=t9	n11	n12	s11=t11	n13	n14
s10=t10	n11	n13	s12=t12	n12	n14
s11=t11	n13	n14			
s12=t12	n12	n14			

4.1. Combinational gates

The algorithm will be explained with the help of the CCC shown in Figure 4. Each channel component is analyzed sequentially by the algorithm. All switch-level simulation algorithms [8] analyze circuits using such partioning schemes, and most model generation algorithms manipulate the resulting equations from such a symbolic analyzer [11, 9]. However, as evident from Figure 4, a nand gate and an inverter becomes part of the CCC, and a straightforward use of the result of such symbolic analysers would hinder the recognition of such combinational gates. If these combinational gates are not recoginzed, it increases the number of tristate devices and the quality of the model is deteriorated.

Each CCC can be thought of as a switch graph, where transistors represent switches connecting circuit nodes. Each switch has a label, which represents the condition under which it conducts. Initially, these conditions represent the “on” condition at the gate terminal (logic 1/0 for N/P transistor). For each switch graph in the circuit, AND-OR reductions are carried out. An AND reduction replaces two transistors in series when there are no other connections at the intermediate node. An OR reduction is carried out whenever there are two switches in parallel.

Initially, the switch graph has a switch for each transistor in the channel connected component. A pseudo code for this portion of the algorithm is shown below:

```
initialize a switch for each transistor
do OR pass
while (true) do {
        if AND pass fails break
        if OR pass fails break
 }
 for each node with switch to power and gnd {
      check for complementarity
 }
```

AND-OR reduction continues till either of these steps fail. For the circuit of Figure 4, Table 1 shows the switch graph, both before and after the reduction. Node numbers and transistor identifiers have been used in the table. As evident from the table, transistors pairs (t1,t2) and (t3,t4) are reduced by an OR and AND step respectively. Each circuit node is then checked for the presense of two switches, one to power and the other to ground. If the Boolean enabling conditions are complementary, a combinational gate has been detected. A BDD package [5] may be used for this purpose. In Table 1, the enabling conditions for the pairs (s13,s14) and (s5,s6) are complementary, and transistors comprising these switches are modeled as combinational gates or inverters. These transistors are removed from further modeling analysis.

4.2. Tristatable drivers

Transistors that fail to be modeled as combinational gates, are often tristatable bus drivers. It is assumed that pull-up and pull-down functions (to power and ground), similar to the ones obtained from AND-OR reduction, are available at each node. It is also assumed that all combinational gates in the circuit have already been found by the previous step. Nodes that are not outputs of combinational gates are analyzed using the algorithm described in this section.

The pullup and pulldown functions at each node (non-combinational) are first decomposed into disjuncts. The gate terminals that comprise each disjunct are analyzed. As an example, consider the circuit of Figure 1. It has one pullup disjunct and three pulldown disjuncts. All gate terminals that appear in the pullup and pulldown disjuncts are traced backwards through combinational gates, till a common fanin cone is found. During this backward trace, if a node that is neither an output of a combinational gate, nor a primary input, Pullup function (PU)

$d1d2 \rightarrow$ $c1c2 \downarrow$	00	01	10	11	$\exists(d)PU$
00	0	1	0	1	1
01	0	0	0	0	0
10	0	0	0	0	0
11	0	1	0	1	1
$\exists(c)PU$	0	1	0	1	

Pulldown function (PD)

$d1d2 \rightarrow$ $c1c2 \downarrow$	00	01	10	11	$\exists(d)PD$
00	0	1	0	1	1
01	0	0	0	0	0
10	0	0	0	0	0
11	0	1	0	1	1
$\exists(c)PU$	1	0	1	0	

Table 2: Example of a bus driver

is reached, the backward cone search process is terminated at that node. Once a cone is found, this procedure guarantees that the pullup and pulldown functions can be expressed as Boolean functions of nodes that are input to the cone. For the circuit of Figure 1, the pullup function has one disjunct, whose input cone can be traced to the signals a, b, c and e, assuming signal $\overline{e}$ can be traced to e. The pulldown function has three disjuncts, and the input cones consist of signal sets {a, e}, {b, e} and {c, e} respectively. The pullup function has only one disjunct {a, b, c, e}.

A subset of the pullup disjuncts is matched with a subset of pulldown disjuncts. For each pair of subsets of pullup and pulldown disjuncts, the input variables are separated into two categories:

- Control Variables: These have the same parity in both subsets.
- Data Variables: These have opposite parity in the two subsets.

For our example, the subsets of pullup disjuncts is {{e, a, b, c}}, while the matching pulldown subsets consist of $\{\{e, \overline{a}\}, \{e, \overline{b}\}, \{\overline{e}, \overline{c}\}\}$. Since e has same parity in both pullup and pulldown disjuncts, it is classified as a control variable. The other variables (a, b and c) are data variables. If a variable appears in only the pullup disjuncts, then that particular subset pair cannot be matched. For example, the pulldown subset $\{\{\overline{a}, e\}, \{\overline{b}, e\}\}$ does not match the pullup subset $\{\{e, a, b, c\}\}$ due to the additional signal c in the pullup subset. In other words, the fanin cone for these subsets do not match.

Once matching subsets of pullup and pulldown disjuncts are found, two tests are carried out for each data and control variable. Since control variables enable the tristate driver, certain assignment(s) to them would cause the pullup and pulldown disjuncts to have opposite values (depending on the values of data variables). These assignments correspond to bus enabling conditions for control variables. For the other remaining assignments, all pullup and pulldown disjuncts evaluate to 0, and the bus is disabled. This condition can be verified both with a truth table, and symbolically with Boolean functions. Table 2 shows an example with two data and two control variables. When the control variables are 01 or 10, both the pullup and pulldown functions are 0. For the other two assignments (00 and 11), the pullup and pulldown functions are complementary to each other.

The above property is equivalent to the following two conditions:

- *Complementary Data Drive:* When all control variables are assigned specific bus enabling values, the pullup and pulldown functions are complementary to each other.
- *Simultaneous Enable Condition:* Both the pullup and the pulldown functions are enabled for the same values of the control variables.

The way to check the first condition symbolically is to evaluate the Existential Quantification of the pullup and pulldown functions with respect to the control variable and checking for complementarity. The value of this operation for the truth table in Figure 2 is also shown in that figure, as the last row and denoted by $\exists(c)PU/PD$. The check for complementarity is trivially done with the help of a BDD package. The second condition requires existential quantification with respect to data variables, and is shown as the last columns for both the pullup and pulldown functions in Table 2, and denoted by $\exists(d)PU/PD$. These conditions are sufficient but not necessary. The sufficiency of these conditions will be explained in a complete version of this paper. As a corollary, it is easy to prove that the rows of the truth table, with the control variables enumerated as rows (as in Figure 2), are either identically zero (bus disabled), or are equal to one other unique combination. This implies that the tristate output depends only on the data variables when the tristate is enabled. A formal proof of this has not been included here due to lack of space, but is not difficult to show. If both the *Complementary Data Drive* and *Simultaneous Enable* conditions are satisfied, the classification into control and data signals is considered to be correct and a tristatable bus driver model like that of Figure 1(c) is generated.

4.3. Multiplexers

A multiplexer model is attempted whenever a node is driven by specific types of gates that have been recognized previ-

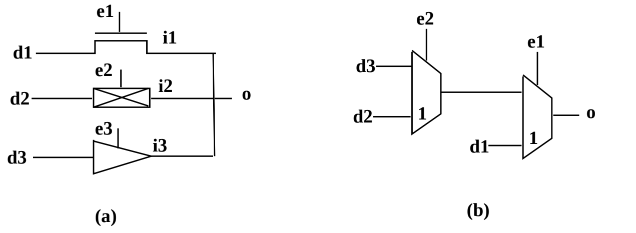

Figure 5: Circuit configurations for multiplexer model

ously. Such driving gates can be any one of the following types: tristatable drivers, pass gates and unidirectional transistors. This is shown in Figure 5(a), where $e1$, $e2$ and $e3$ are potential select signals for the multiplexer branches, and $d1$, $d2$ and $d3$ are data signals. Note that transistor $i1$ needs to have the proper direction to be considered a branch of the multiplexer. If all subsequent tests pass, the model generated may be similar to the one shown in Figure 5(b).

A transistor direction assignment algorithm is helpful for correct model generation. The reader is referred to the literature for details [14]. We mention some of the relevant details in the context of the example of Figure 2. Unmodeled transistors that are either driven by primary inputs or outputs of combinational gates, including multiplexers, are always directed away from the driven node. Hence, the recognition of multiplexers aid the process of transistor direction assignment.

Considering the example of Figure 2(a), since nodes d and $\overline{d}$ are primary input and inverter output respectively, transistors $t9$, $t10$, $t11$ and $t12$ are directed as shown in the diagram. Transistors $t9$ and $t10$ are then inferred to be a 2-input multiplexer driving node $n3$. This algorithm is described later in this section. A similar reasoning applies to transistors $t11$ and $t12$ and node $n4$. After this initial recognition, the intermediate circuit model is shown in Figure 2(d). This procedure is repeated twice and an intermediate model is shown in Figure 2(e). The final model is shown in Figure 2(f). This model was possible only after transistor direction assignment at intermediate stages of the model generation algorithm.

Control signals like $e1$, $e2$ and $e3$ of Figure 5 have to satisfy additional constraints in order to qualify for multiplexer control points. These signals have to be mutually exclusive and one of them has to evaluate to a logical 1 at all times:

- mutual exclusion : $\forall i, j, i \neq j, e_i \bigwedge e_j = false$
- always driven : $\bigvee e_i = true$

Evaluation of these conditions is performed with the aid of Binary Decision Diagrams (BDDs). To keep the size of BDDs small, they are build incrementally, such that the Boolean variables used in the BDD construction are internal signals in the circuit (as opposed to primary inputs and latch outputs). BDDs for cones build from inputs to a combinational frame would not work for two reasons:

```
SetOfNodes InitAllDrvs(Node $e_i$, Mxdpth l){
  if (l = 0) return $\emptyset$;
  $d_i$ = Backtrace($e_i$); // inverter collapsing
  AllDrvs = { $d_i$ }
  if (UniqueComb($d_i$) = $\emptyset$)
    return AllDrvs;
  for each input $e_{i+1,j}$ of UniqueComb($d_i$)
    AllDrvs=AllDrvs $\bigcup$ InitAllDrvs($e_{i+1,j}$,l-1);
  return AllDrvs;
}

Boolean CheckAllDrvs(Node $e_i$, Limit l,
                          SetOfNodes firstCone) {
  if (l = 0) return 0;
  $d_i$ = Backtrace($e_i$); // inverter collapsing
  if ($d_i \in$ firstCone) return 1;
  if (UniqueComb($d_i$) = $\emptyset$) return 0;
  for each input $e_{i+1,j}$ of UniqueComb($d_i$)
    if (!CheckAllDrvs($e_{i+1,j}$, l-1, firstCone))
        return 0;
  return 1;
}

Boolean CheckMux(Limit l, Nodes $e_1, \ldots, e_N$) {
  firstCone = InitAllDrvs($e_1$, l);
  for all nodes in $\{e_2, e_3, \ldots, e_N\}$
    if (!CheckAllDrvs($e_i$, l, firstCone)) return 0;
  build BDD $f_i$ for $e_i$ for i $\in \{1, \ldots, N\}$
 // check exclusivity and always driven cond
  for each $e_i, e_j, i \neq j$
    if ($f_i \bigwedge f_j \neq \emptyset$) return 0;
  if ($\bigcup_i f_i \neq 1$) return 0;
  return 1;
}
```

Figure 6: Multiplexer Extraction Pseudo Code

1. The size of BDDs are usually too big for many circuits of practical interest.
2. The combinational frame boundary is not known a priori.

The BDD based analysis is presented in Figure 6. First, a limited depth input cone for node e_1, the first branch select signal, is built. The inverter chain driving e_1 is collapsed and node d_1 driving this inverter chain is found. This inverter collapsing procedure is called Backtrace in Figure 6. Note that note d_1 may be identical to e_1. The depth of d_1 is set to 1. The static complementary gate driving d_1, if any, is then analyzed. If no unique static complementary gate drives d_1, this backtrace procedure stops. Otherwise, it continues at all inputs to this static complementary gate, denoted $e_{21}, e_{22}, ..., e_{2_k}$, with the depth parameter incremented to 2. The inverter chain driving each of the nodes, e_{2i}, is traversed, and the unique static combinational driver to the chain rooted

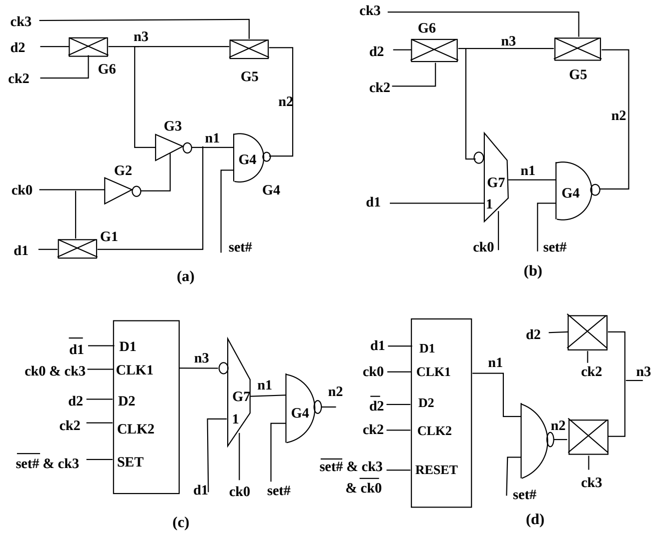

Figure 7: Latch example

at d_{2i}, is found. As noted earlier, node d_{2i} may be the same as node e_{2i}. The procedure is then continued for each node d_{2i}, until either the depth limit is reached, or node d_{2i} is not driven by an unique static complementary gate. The drivers of these inverter chains, $d_1, d_{21}, d_{22}, ..., d_{2i}, ...$ are all stored in an array for later use. Evaluation of all the drivers is shown in procedure InitAllDrivers in Figure 6. Once this backward search terminates for node e_1, a similar search is initiated for each of the remaining nodes e_k. However, these subsequent searches are terminated at a driver d_{kj} whenever d_{kj} is a node stored as a result of the first search (for node e_1). If the depth limit is reached for any of $e_2, e_3, ..., e_N$, then the input cone for these nodes are not the same within the maximum depth limit set. To continue the search, the depth limit is incremented and the process is repeated. This cone matching procedure is called CheckAllDrivers in Figure 6. Either a cone that drives all the enable signals are eventually found, or the process terminates without success. If a matching cone is found, the BDDs are evaluated and the multiplexer conditions are checked. The BDDs are evaluated in terms of the driver nodes that lie at the boundary of the cone (as opposed to drivers internals to the cone). As mentioned previously, the MUX conditions check for pairwise mutual exclusivity and one of the conditions must always be true. The main procedure is called CheckMux in Figure 6.

5. Sequential elements

This algorithm will be explained with reference to the circuit shown in Figure 7(a). The reader can verify the model shown in Figure 7(b) can be obtained using ideas presented in Section 4.3. For this circuit, several latch models are possible for the latter circuit, as explained in the following paragraphs.

```
(1) Find a simple cycle of nodes $n_1, \ldots, n_N$, driven by
    gates $G_1, \ldots, G_N$.
(2) For every pair of nodes $(n_i, n_j), i \neq j$, evaluate
    parity($i, j$) and prop($i, j$).
(3) Choose some node $n_k$ state node $SN$.
(4) For each combinational gate $G_k$ on cycle
      For each off-path input input $n_{ki}$
        let $c_{ki}$ = controlling value
        let $p_{ki}$ = parity of $G_k$
        if $c_{ki} \otimes p_{ki} \otimes parity(k, SN) = 1$
          set input = $(n_{k,i} \otimes p_{ki}) \bigwedge prop(k, SN)$
          else reset = $(n_{k,i} \otimes p_{ki}) \bigwedge prop(k, SN)$
(5) For each mux gate $G_k$ on cycle
      For each off-path input input $n_{ki}$
        let $mp_{ki}$ = mux input parity
        let $ms_{ki}$ = mux branch select for $n_{ki}$
        instantiate new port for latch
          data = $n_{ki}$, clock = $ms_{ki} \bigwedge prop(k, SN)$
          data parity = $mp_{ki} \otimes parity(k, SN)$
(6) For each offpath passgate $G_k$ that drives $n_j$ in loop
      For pass gate input $n_{ki}$ and enable $e_{kj}$
        instantiate new port for latch
          data = $n_{ki}$, data parity = parity(j,SN)
          clock = prop(j,SN) $\bigwedge e_{kj}$
(7) Write out set, reset, and data and clock signals for
    each port of latch, and all gates in circuit except on-
    path gate $G_k$ driving state node $SN$.
```

Figure 8: Latch Extraction Pseudo Code

A pseudo code in presented in Figure 8. The first step consists of detecting simple loops in the circuit. For the example shown, such a loop consists of nodes $(n1, n2, n3)$, which are outputs of gates $(G7, G4, G5)$ respectively. These gates are the *on-path* gates, as opposed to $G6$, which is off-path, with respect to the loop detected. Similarly, input $d1$ of multiplexer $G7$ is an off-path input. For each pair of distinct nodes in the cycle, inversion parity along the cyclic path is also evaluated. The parity calculation is commutative, because net parity along the loop has to be zero. As an example, parity($n1, n2$) is 1. Of the three possible choices, one of the on-path nodes that lie on the cycle is designated as a state node (referred to as SN in Figure 8). Additionally, a propagation condition, $prop(n_i, n_j)$, is evaluated for each pair of nodes in the loop. It represents the enabling condition for signal flow from node n_i to n_j .

The next few steps (Steps 4-6) analyze how the logic values on the loops can be changed by the surrounding logic and off-path inputs to gates that lie on the loop (on-path gates). For combinational on-path gates, these offpath inputs are modeled as set/reset inputs to the latch. The output values of on-path combinational gates can be changed by

Table 3: Modeling and ATPG Results

		Non-mux version					Mux version				
(1) ckt	(2) trans	(3) gates	(4) muxes	(5) faults	(6) eff (%)	(7) cov (%)	(8) gates	(9) muxes	(10) faults	(11) eff (%)	(12) cov (%)
ckt1	11335	9835	226	11134	99.9	81.5	9423	290	10494	99.9	87.6
ckt2	15432	13420	410	18332	99.6	77.3	11988	612	15086	99.5	86.0
ckt3	19838	8530	228	10629	98.3	78.9	6230	345	9736	98.9	92.2
ckt4	23819	22881	912	29817	79.7	46.2	20047	1002	24105	79.3	70.6
ckt5	26747	21273	625	17623	99.6	79.5	23629	773	22387	99.9	93.2
ckt6	41654	49873	2484	95973	35.6	23.5	25115	4094	56678	90.9	73.0
ckt7	49798	52884	534	58332	99.8	85.5	46178	970	53173	99.6	89.0

controlling values at the off-path inputs to these gates. For the example circuit, the node $set\#$ can force node $n2$ to 1, and hence behaves as an active low set input, provided $n2$ is the state node of the latch. However, if the propagation of values from the output of the combinational gate to the state node depends on values of enabling signals along the path, this off-path input, in conjunction (logical AND) with appropriate clocking is modeled as a synchronous set/reset port. For the node $set\#$, assume node $n3$ is the state node. Since propagation to $n3$ depends on $ck3$ (the propagation enabling signal condition), a synchronous set signal ($\overline{set\#} \bigwedge ck3$) will be created.

For on-path multiplexer gates, the analysis is similar, except that new data ports are instantiated. The multiplexer branch select signal for the off-path input, in logical conjunction with the propagate signals (from multiplexer output to state node), constitute the clock signal for the port. The propagation parity from the multiplexer output to the state node, and the multiplexer input data parity constitute the parity for the data signal for the instantiated port. For the example circuit, if $n3$ is the state node, the clock signals would be $ck0 \bigwedge ck3$. The data signals for the port would be $\overline{d1}$, due to the inverting parity of $G4$ ($SN = n3, prop(n1, n3) = 1$). Analysis for other gate types (transistors, tristatable drivers etc.) are similar, and the algorithm for transmission gates only is shown in Step 6 of Figure 8.

Finally, a latch block, with the new data ports and/or set/reset inputs, with all its surrounding gates, is written out in a netlist file. The only exception is the on-path gate whose output node was designated as the state node. For our example, Figure 7(c) shows a model with $n3$ chosen as a state node. Figure 7(d) shows another model with $n1$ as a state node. Note that $\overline{set\#}$ becomes a synchronous reset input in this model.

6. Results

A number of datapath circuits with pass transistor logic were used for benchmarking. Experimental verification and comparison with the only publicly available modeling tool [9] was difficult because of its inability to model sequential logic in a manner that can be used directly by an ATPG tool. Hence, our own algorithm was used to generate two models of the same circuit. The multiplexer extraction procedure of Section 4.3 was turned off while generating one of the two models. A commercially available ATPG tool was then used to generate patterns for both models.

All ATPG tools have learning capabilities and some of the multiplexers left out in the first model were recognized by the ATPG tool before starting pattern generation. Tristates that are detected as multiplexers during the learning stage help in classifying some faults as provably untestable. Otherwise, these faults remain untested and lower subsequent fault coverage. Table 3 data reports the number of multiplexers that our algorithm was able to find, and this is compared to those found by the ATPG tool. Some of the multiplexers were recognized by the ATPG tool as Exor and Exnor gates, and these gates were added in the ATPG tool data. Along with fault coverage, fault efficiency (found by removing the provably untestable faults) numbers for both models were also obtained. Both models were verified for correctness.

Referring to Table 3, Column 2 lists the number of transistors in each circuit. Columns 3-7 list ATPG statistics for the model with multiplexer detection suppressed, while Columns 8-12 list the similar data when all the algorithms outlined in the paper are included. Column 4 lists the number of multiplexers that were found by the ATPG tool during the learning phase, while Column 9 lists this same number using our algorithm. The fault efficiency numbers reported in columns 6 and 11 were obtained from the ATPG tool, after removing the provably untestable faults from consideration.

As is evident from Table 3, the difference in fault coverage between the two models is directly correlated to the number of multiplexers that were left unmodeled by both our program, and the learning stage of the ATPG tool. The result of this inefficiency contributes an average decrease in fault coverage of about 15%-20%. This figure was as high as 49% for $ckt6$. For the 4100 multiplexers in the circuit, about 2500 could be inferred by the ATPG tool. Further examination of the model revealed several parity circuits of the type shown in Figure 2. These were either unmodeled or partially modeled by the ATPG tool and contributed to the sharp decrease in fault coverage. Our implementation uses an iterative algorithm, wherein transistors are first aligned and the multiplexer algorithm applied at each node, till no additional multiplexers are found in an iteration. At the end of each iteration, based on the new multiplexers that were found, additional transistors driven by these multiplexer outputs are aligned. The whole process is then repeated. The multiplexer structures in $ckt6$ were 6 levels deep, requiring seven iterations of the algorithm. The buses in the circuit are 32 bits wide.

7. Conclusion

The work presented in this paper addresses many issues related to modeling and ATPG of custom digital circuits. While the amount of literature related to ATPG research is substantial, there is further scope for improvement in issues related to modeling. Modeling has a direct impact on ATPG. Though these algorithms were presented in the context of ATPG and fault simulation, their application to other areas of CAD, including cell characterization, logic simulation and formal verification, is a logical extension.

8. Acknowledgements

The author would like to thank V. Gangaram of Intel Corporation for managerial assistance and for running some of the ATPG tools for the sample circuits, and V. D. Agrawal of Agere Systems for comments on an earlier version of the paper.

References

[1] P. Agrawal and V. D. Agrawal. Can Logic Simulators Handle Bidirectionality and Charge Sharing. In *Proc. International Symposium on Circuits and Systems (ISCAS)*, pages 411–414, May 1990.

[2] P. Agrawal, S. H. Robinson, and T. G. Szymanski. Automatic Modeling of Switch-Level Networks Using Partial Orders. *IEEE Trans. Computer-Aided Design*, 9(7):696–707, July 1990.

[3] S. Bose, V. D. Agrawal, and T. G. Szymanski. Algorithms for Switch Level Delay Fault Simulation. In *Proc. International Test Conf.*, pages 982–991, 1997.

[4] R. E. Bryant. A Switch-Level Model and Simulator for MOS Digital Systems. *IEEE Trans. Computers*, C-33(2):160–177, Feb. 1984.

[5] R. E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. In *IEEE Transactions on Computers*, volume C-35, 8, pages 667–691, August 1986.

[6] R. E. Bryant. Algorithmic Aspects of Symbolic Switch Network Analysis. *IEEE Trans. Computer-Aided Design*, CAD-6(4):618–633, July 1987.

[7] R. E. Bryant. Boolean Analysis of MOS Circuits. *IEEE Trans. Computer-Aided Design*, CAD-6(4):634–649, July 1987.

[8] R. E. Bryant. A Survey of Switch-Level Algorithms. In *IEEE Design and Test of Computers*, volume 4, pages 26–40, August 1987.

[9] R. E. Bryant. Extraction of Gate Level Models from Transistor Circuits by Four Valued Symbolic Analysis. In *Internation Conference on Computer Aided Design*, pages 350–353, November 1991.

[10] K. L. Einspahr and S. C. Seth. A Switch-Level Test Generation System for Synchronous and Asynchronous Circuits. *J. Electronic Testing: Theory and Applications*, 6(1):59–73, Feb. 1995.

[11] A. Jain and R. E. Bryant. Mapping Switch Level Simulation onto Gate Level Hardware Acceleators. In *Design Automation Conference*, pages 219 – 222, June 1991.

[12] S. K. Jain and V. D. Agrawal. Modeling and Test Generation Algorithms for MOS Circuits. *IEEE Trans. Computers*, C-34(5):426–433, May 1985.

[13] K. J. Lee, C. A. Njinda, and M. A. Breuer. A Switch-Level Test Generation System for CMOS Combinatinal Circuits. *IEEE Trans. Computer-Aided Design*, 13(5):625–637, May 1994.

[14] K. J. Lee, C. N. Wang, R. Gupta, and M. A. Bruer. An Integrated System for Assigning Signal Flow Directions to CMOS VLSI. In *IEEE Transactions on Computer Aided Design*, volume 14, No 12, pages 1445–1458, December 1995.

[15] M. D. Schuster and R. E. Bryant. Concurrent Fault Simulation of Digital MOS Circuits. In P. Penfield, Jr., editor, *Proc. Conf. on Advanced Research in VLSI*, pages 129–138, Jan. 1984.

[16] R. L. Wadsack. Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits. *Bell System Tech. J.*, 57(5):1449–1474, May-June 1978.