Extraction of Schematic Array Models for Memory Circuits

Soumitra Bose, Amit Nandi*
Test Technology Group, Intel Corporation, Folsom, CA 95630

Abstract

The modeling and simulation of memory circuits remains an outstanding problem when accuracy with respect to the actual schematic implementation is desired. Functionally equivalent RTL models often cannot be used for designs with embedded memory blocks, because schematic models for the surrounding logic may be required for fault modeling accuracy. Existing methods derive a latch model that essentially represents each memory location as a latch primitive, and have a large number of gates. We present new algorithms that model such circuits as decoded *arrays that access entire rows of cells for individual read and write operations. Decoded array models allow fault modeling accuracy for the surrounding logic, including the memory address decoder. Experimental data show improvements of an order of magnitude for both logic and fault simulations, when compared to the equivalent latch model.*

1 Introduction

Extracted gate level models [1, 2, 3] may be used for logic simulation, formal verification, or test related problems like fault simulation and test generation. However, these techniques are unable to model memory circuits in a satisfactory manner. For such circuits, the Channel Connected Sub-Networks (CCSN) containing the state elements are very large, and correct node and transistor size assignments are required for the resulting switch-level model to simulate correctly. Moreover, these approaches generate a model that is not as useful for test problems like ATPG. This is due to many peripheral circuit elements, like combinational gates, that are not directly related to the memory functionality, but indeed belong to the same CCSN containing the state elements. The functionality of such combinational gates gets absorbed into the equations modeling the memory bank, which makes tasks like backward justification and forward propagation almost impossible to perform with any degree of efficiency, both in terms of execution times and fault coverage.

Perhaps, the most extensive work on memory array modeling can be found in the formal verification literature [4, 5]. These methods require phase-accurate knowledge of circuit behavior. The specification itself may require knowledge of the address decoders [5], which are often impossible to isolate without detailed design knowledge. These techniques usually represent memory contents symbolically in terms of address and data values. Hence, their use is restricted to verifying properties that require analysis of a few consecutive cycles only. Furthermore, the address decoders are usually retimed using sequential and domino logic and often lie in some other circuit that is modeled separately. The non-availability of the decoder and the need for the phase-accurate specification model makes algorithms similar to the *Efficient Memory Model* [5], virtually unusable for test problems like fault simulation.

For fault simulation and test generation, the only proven method consists of inserting behavioral models extracted from a high level language like Verilog or VHDL [6]. However, this approach cannot be used when address decoders are designed separately in another unit. This paper presents a new one-hot decoded model that does not require phase accurate specification, avoids the need for memory intensive symbolic simulation of arrays, and can be used without design specific knowledge of address decoders.

Table 1: Model sizes for datapath circuits

circuit	trans	gates	efficiency
d1	19641	7820	2.51
d2	41521	16775	2.48
d3	23658	9634	2.46
d4	24417	11254	2.17
d5	64713	26866	2.41
m1	77112	108423	0.71
m2	86915	90943	0.96
m3	122005	150090	0.81
m4	388978	477007	0.82
m5	901042	1229080	0.73

2 Motivation

Figure 1(a) shows a column of a memory bank with precharged bit lines and a sense amplifier that functions as a read port. To write a cell, one of the bit lines $bl1$ and $bl2$ is discharged, and the word lines, $wl1$ and $wl2$, are then asserted. A read operation proceeds by asserting simultaneously a word line, the read enables lines $rd1$ and $rd2$, and the sense amplifier enable line e. A gate-level latch model for the above design is shown in Figure 1(b). The feedback

*email:{Soumitra.Bose,Amit.Nandi}@intel.com

1530-1591/04 $20.00 (c) 2004 IEEE

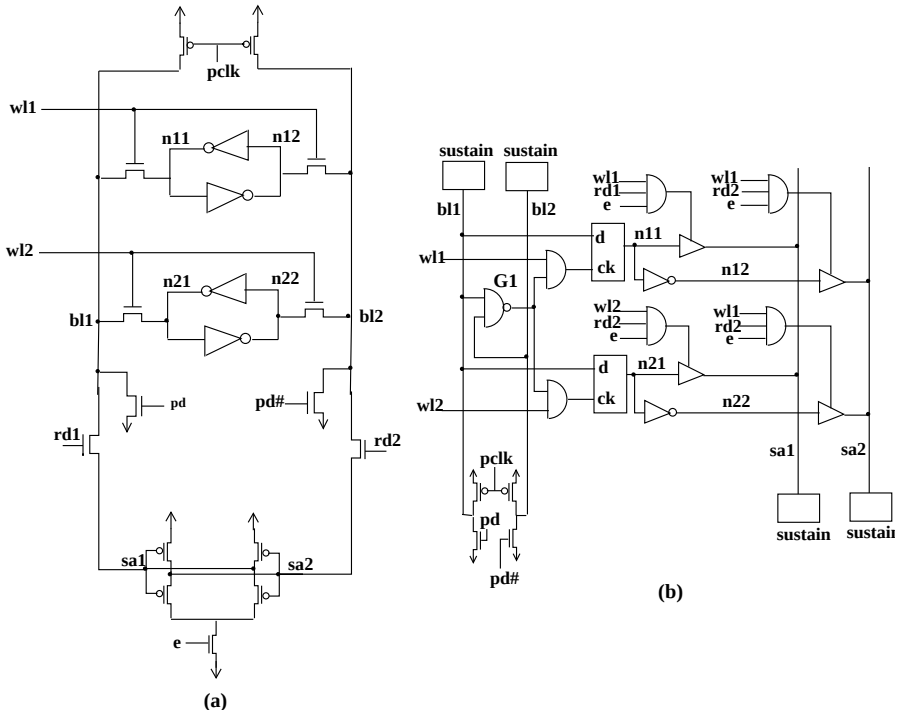


Figure 1: Model Size Explosion for Memory Circuits

configuration in the sense amplifier is modeled by the logic sustainers at nodes $sa1$ and $sa2$. The precharge phase for the bit lines (when precharge clock $pclk$ is asserted low) is different from the phase when the word lines are asserted. The logic sustaining elements at nodes $bl1$ and $bl2$ allows these nodes to stay at their precharged values till the phase they are conditionally discharged by signals pd and $pd\#$.

The above design has six transistors per memory cell. It also requires six gate elements to model each cell. The top half of Table 1 presents model size for several datapath circuits, where the typical ratio of the number of transistors to the number of gate elements, referred to as model efficiency, is observed to lie between 2 and 3. The lower half of Table 1 shows these same numbers for memory circuits, where this ratio is observed to fall below one. Reducing the size of the model accelerates both logic and fault simulations. The improvement in simulation runtime is dramatic for data caches and appreciable for register files, and is the result of a significant increase in model efficiency for many classes of memory circuits.

3 Tree Isomorphism

While finding general isomorphic graphs is a "hard" problem, certain special cases are known to be easy. In this section, we present a variant of a tree isomorphism algorithm, originally presented in a classical text [7].

Given two trees $T_1 = (N_1, E_1)$ and $T_2 = (N_2, E_2)$, the algorithm finds an order for the children of each node of

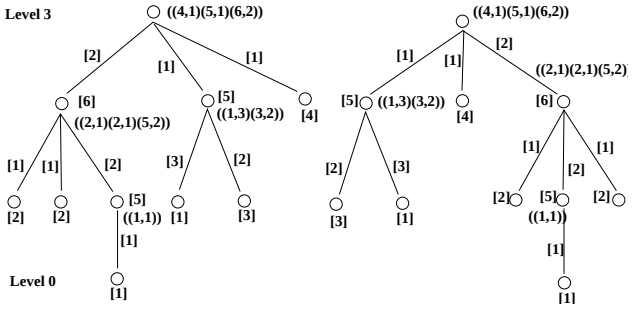


Figure 2: Tree Isomorphism Example

T_2 such that T_1 and T_2 are identical except for the names of the nodes. Unlike usual trees, each node and edge (of $T_1 = (N_1, E_1, IN_1, IE_1)$) has attributes $IN_1 : N_1 \rightarrow I$ and $IE_1 : E_1 \rightarrow I$, where I is the set of integers. We will refer to labelings IN and IE as node and edge incidence counts respectively. The reason for this nomenclature will be evident shortly. All edge labelings and node labelings for leaves of the trees are input to the algorithm. Node labelings for internal nodes are evaluated by the algorithm, as explained below.

For trees $T_1 = (N_1, E_1, IN_1, IE_1)$ and $T_2 = (N_2, E_2, IN_2, IE_2)$, the algorithm can be summarized by the following steps:

Algorithm 1:

1. Evaluate for all leaf nodes $n_1 \in N_1$ and $n_2 \in N_2$, $M_1 =$

$Max(IN_1(n_1))$, and $M_2 = Max(IN_2(n_2))$. If $M_1 \neq M_2$, exit (the trees are not isomorphic).

2. Order all nodes of T_1 and T_2 by increasing levels.
3. Assume L_1 (L_2) is the list of nodes of T_1 (T_2) at level i. Evaluate an integer, Id, at each node in L_1 and L_2 by following steps (4)-(8). Assume Id is evaluated for each node at level $(n-1)$ and that the nodes at level $(i-1)$ are sorted by increasing order of Id.
4. If node $n_1 \in L_1$ is a leaf, $Id(n_1) = IN_1(n_1)$. If n_1 is a non-leaf, go to step (5). Similarly, for leaf node $n_2 \in L_2$, $Id(n_2) = IN_2(n_2)$. For non-leaf node n_2, go to step (5).
5. If node n_1 is a non-leaf at level i, assume nodes at level $(i-1)$ have already been evaluated. Let the children of n_1 be the nodes $c_{11}, c_{12}, \ldots, c_{1q}$. Let $e_{11}, e_{12}, \ldots, e_{1q}$ be the respective tree edges from n_1 to $c_{11}, c_{12}, \ldots, c_{1q}$. Assign a list of tuples to n_1 where each member of the list is itself a 2-tuple. Each 2-tuple is derived from a child, c_{1i}, of $n1$ and consists of two elements: $(Id(c_{1i}), IE(e_{1i}))$. The list assigned to node n_1 is
$$\{(Id(c_{11}), IE(e_{11})), \ldots (Id(c_{1q}), IE(e_{1q}))\}$$
6. Sort the lists evaluated at each non-leaf node in list L_1. For the first unique list at non-leaf node $n_1 \in L_1$, assign the integer $(M_1 + 1)$ to $Id(n_1)$. For the second unique list at node $n_1' \in L_1$, assign $(M_1 + 2)$ to $Id(n_1')$, and so on.
7. Repeat Step (6) for non leaf nodes in list L_2.
8. If the list of integers $Id(n_1), n_1 \in L_1$ and $Id(n_2), n2 \in L_2$, evaluated in steps (3)-(6), are not the same, the trees are not isomorphic. If the lists are identical, sort the nodes of L_1 and L_2 using the evaluated Id values from Step (6) and proceed to the next level (i+1) at step (3). If the roots are at level $i+1$, go to step (9).
9. At the roots, evaluate the sorted list of steps (5)-(6). If the lists are identical (non-identical), the trees are isomorphic (non-isomorphic).

Figure 2 shows two trees and the intermediate lists and Id values evaluated at individual nodes. The values of M_1 and M_2 evaluated in Step (1) of the above algorithm is 4. The numbers shown is square brackets next to the nodes are the values of Id, while those next to the edges are values of IE. The lists shown next to non-leaf nodes are those evaluated at step (6) of the above algorithm. The list at the roots of the trees are equal and hence, the trees are isomorphic. This can be verified manually.

Figure 3 shows a small memory circuit, where the analysis of the read logic (transistors gated by signals ren[0] and ren[1]) is assumed to be carried out separately. We show how write logic for these cells can be identified quickly using tree isomorphism. The signal flow graph for this circuit is shown in Figure 4(a), where the eight inverter nodes are assumed to be children of a fictitious root node. The node incidence count for signal vss is 12, while each of the data input signals ($di0$[0], $di1$[0], $di1$[0] and $di1$[1]) have node incidence counts of 2, and edge incidence counts of 2, respectively. The write enable signals $en0$[0], $en0$[1], $en1$[0] and $en1$[1] have edge incidence counts of 4. The set/reset signals have edge incidence counts of 2.

The maximum node incidence count for the circuit in Figure 3 is 12 (due to node vss). The five unique lists evaluated by the tree isomorphism algorithm are shown in Figure 4(a), and are assigned integer identifiers ranging from 13 to 16. For $cell00$ (nodes mc[00] and $mc\#$[00]), the assigned lists are identical to those of $cell10$ (nodes mc[10] and $mc\#$[10]). Once a model for $cell00$ is derived, the model can be reused for $cell10$ by a simple renaming of signals, as explained below.

The total number of signal paths to nodes mc[00] and $mc\#$[00] is five. However, some paths *dominate* others. Path A dominates path B if activating path B implies activating path A, while the reverse is not necessarily true. It can be easily verified that the paths with two transistors to node $mc\#$[00] are dominated by other paths, and can be ignored from further analysis. For the remaining three paths, a dual ported latch with a set input can be derived for signal mc[00], as shown in Figure 4(b). Since the subtrees rooted at mc[10] and $mc\#$[10] are isomorphic to those mc[00] and $mc\#$[00], the model for these nodes can be derived by name substitution. Tree isomorphism is linear in complexity of the size of the tree. However, the size of the tree may be larger than the original graph representation of the circuit. This occurs due to the conversion of the directed acyclic graph to a tree, and requires some nodes and edges to be replicated. In practice, the payoff from a linear complexity algorithm offsets the increase from a larger number of nodes in a signal flow tree.

The overall algorithm can be summarized by the following steps:

1. For each new back-to-back inverter pair found, repeat steps (2)-(5).
2. Initialize a list of transistors to NULL.
3. For the two CCSNs that contain the outputs of an inverter pair, (these two CCSNs may be identical), add all transistors in these two components to the list initialized in Step (2).
4. For each transistor added to the list in step (3), check if its source/drain terminal is the output of a back-to-back inverter pair different from the one found in step (1). For such inverter pair output nodes, go back to step (3).
5. For the list of transistors initialized in steps (2)-(4), evaluate tree isomorphism algorithm at each inverter output.

The above procedure essentially collects all transistors in several CCSNs that share back-to-back inverter pairs and generates models for these memory cells simultaneously, thus avoiding repeated analysis of similar circuit structures.

4 Array Model for Memory

Figure 5 shows a block diagram of the type of array models that are extracted by the algorithm. Each memory block is

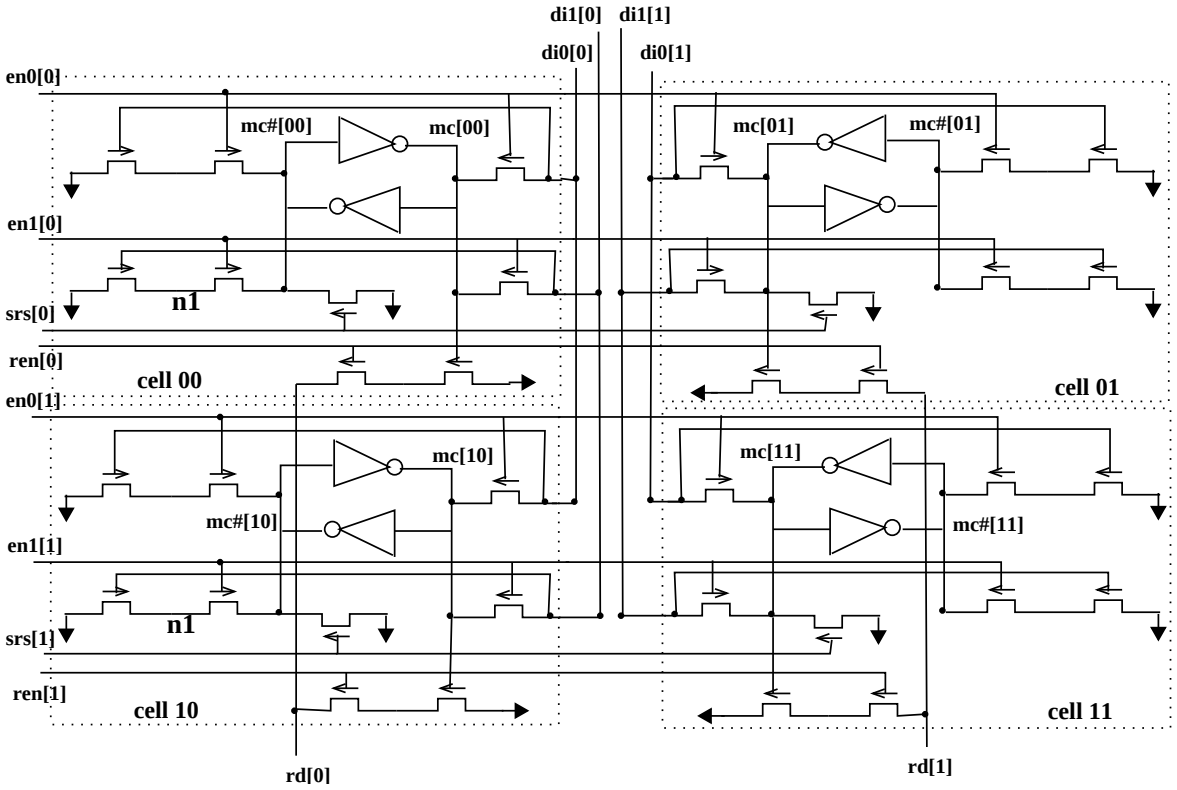
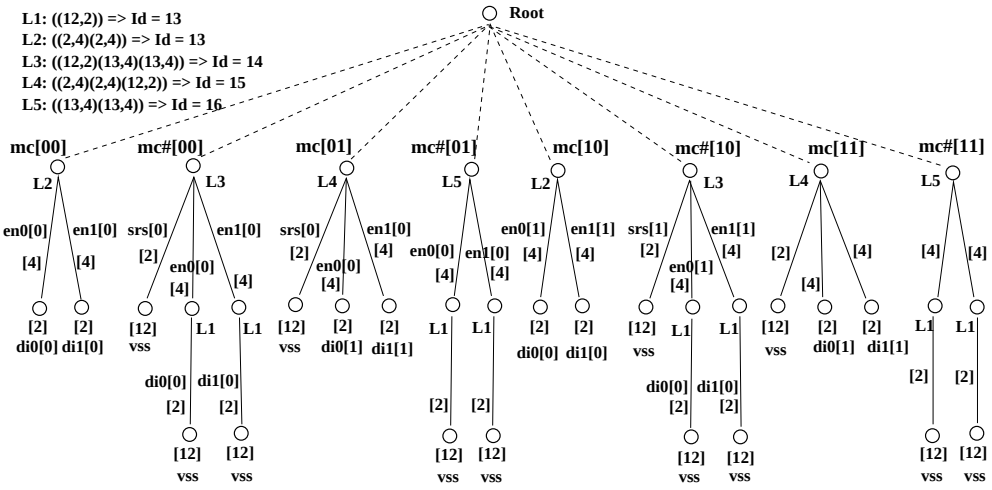
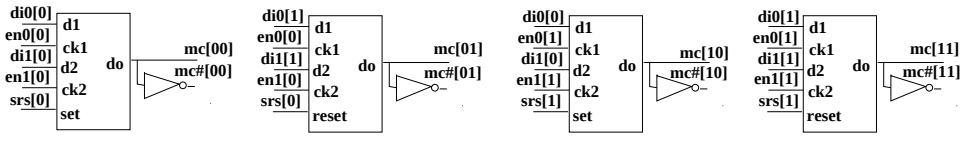


Figure 3: A Dual Ported 2X2 Memory Array



(a) Details for Tree Isomorphism Algorithm



(b) Generated Models for Each Cell

Figure 4: Tree Isomorphism Algorithm for Figure 3

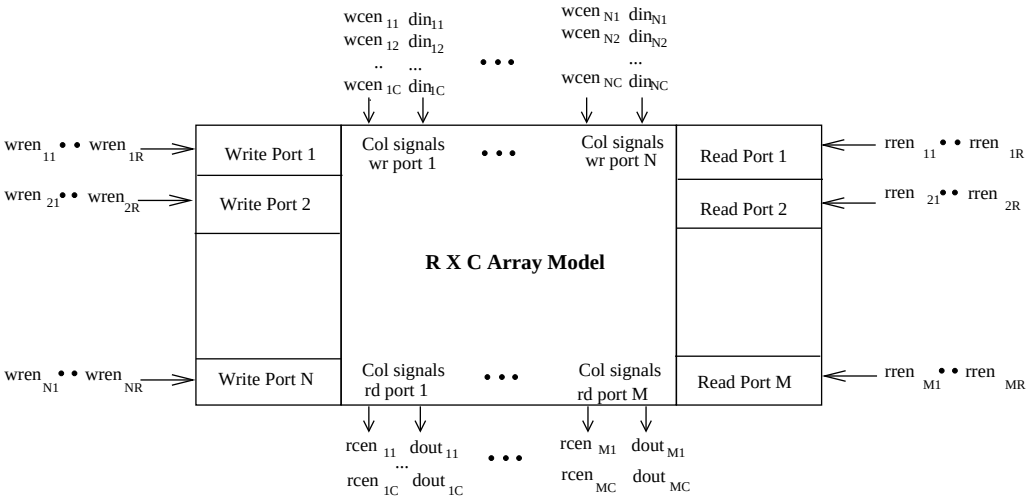


Figure 5: Array Model for Memory

partitioned into R rows and C columns, accessible through N write and M read ports. For the i^{th} write port, there are R row enable signals $(wren_{i1}, wren_{i2}, \ldots, wren_{iR})$, C data input signals $(din_{i1}, din_{i2}, \ldots, din_{iC})$ and C column enable signals $(wcen_{i1}, wcen_{i2}, \ldots, wcen_{iC})$. For the i^{th} read port, there are R row enable signals $(rren_{i1}, rren_{i2}, \ldots, rren_{iR})$, C data output signals $(dout_{i1}, dout_{i2}, \ldots, dout_{iC})$ and C column enable signals $(rcen_{i1}, rcen_{i2}, \ldots, rcen_{iC})$. When the k^{th} row enable signal for the i^{th} write port $(wren_{ik})$ is high, the j^{th} cell belonging to that row in location (k, j) is written provided the column enable signal $wcen_{ij}$ is also high. The data written is the logic value of signal din_{ij}. The behavior for the read ports is analogous. The content of the memory bank at location (k, j) is read onto data output node $dout_{ij}$ for port i provided row enable signal $rren_{ik}$ and column enable signal $rcen_{ij}$ are both high. For many memory banks, the column enable signals for a specific port are often trivially tied to the logic 1 state. For such ports, the rows are written (or read from) in blocks of size C, where C is the number of columns in the block.

Once a latch model similar to the one shown in Figure 4(b) has been extracted, an array model can be obtained quite easily. The only additional step necessary at this stage is the identification of the read logic. We have assumed that the read logic is one of two types: (1) domino gates where transistors in the pulldown logic are gated by memory cell contents or their complements or (2) sense amplifier structures similar to the one shown in Figure 1(a). For the 2x2 array example of Figure 3, the read enable signals are identified as $en[0]$ and $en[1]$. The set/reset signals are treated as write ports with vcc and vss as data signals, and $srs[0]$ and $srs[1]$ as write enable signals.

Once a latch model has been extracted for each memory cell, all latches are sorted lexicographically in increasing order (all signals are associated with unique integer identifiers) of (1) write enable, (2) read enable, (3) write data and (4) read data signals. This order of sorting places all cells that are modeled in the same row of an array in consecutive positions. Latches in the same row can now be partitioned into separate columns, after comparison with other rows that have identical data signals.

5 Results

Table 2 presents results obtained for several memory circuits. The fault simulator can support array primitives like that of Section 4. Column 2 shows the number of transistors, while column 3 shows the number of memory cells in each of these circuits. Column 4 shows the number of gates required to model these memory circuits using latch primitives, while column 5 shows the number of gates when array models were extracted using the algorithm of Section 4. Each array primitive is counted as a single primitive because storage requirement for an array primitive is negligible. Column 6 shows the reduction in the model size. For logic simulation, columns 8 and 9 show execution times for various vector sets on a HP Unix workstation. Column 10 shows the speedup obtained as a result of array modeling. As evident from this table, the acceleration obtained for logic simulation is substantial for these designs.

Columns (11)-(14) of Table 2 also tabulate results when these circuits were simulated with a varying number of faults. The number of faults was chosen to keep execution times within reasonable limits. Several observations can be made regarding these results. Any speedup observed during logic simulation was observed during fault simulation also. However, the speedup observed during fault simulation was usually lower than that of logic simulation, with one exception. For circuit $m8$, the opposite was true. Hyperactive faults were detected, and subsequently dropped, during simulation of the array model only. These faults were simulated throughout the entire sequence in the latch model.

Excluding $m8$, speedup factors were observed to be lower for fault simulation. The fault simulator uses a dif-

Table 2: Simulation Results for Array Models

			Model Size Reduction			Logic Sim Results				Fault Sim Results			
(1) ckt	(2) trans	(3) cells	(4) latch	(5) array	(6) comp	(7) vec	(8) latch	(9) array	(10) speed	(11) cov(%)	(12) latch	(13) array	(14) speed
m1	40.5	4100	73.5	7.3	10.1	v1	77	41	1.9	37.1	1505	811	1.9
m2	48.4	4800	28.0	5.5	5.09	v1	123	60	2.05	73.4	1269	933	1.4
						v2	124	61	2.03	61.0	1393	944	1.5
m3	60.8	6480	32.0	8.8	3.64	v1	117	62	1.89	75.9	1486	920	1.6
						v2	106	57	1.86	66.8	1712	908	1.9
m4	77.1	7168	108.4	9.5	11.41	v1	750	60	12.5	25.0	3743	688	5.4
						v2	839	59	14.22	48.8	3901	675	5.8
						v3	787	83	9.48	32.2	5305	891	6.0
m5	86.9	9216	90.9	9.7	9.37	v1	65	39	1.67	41.2	646	450	1.4
						v2	67	39	1.72	46.5	680	441	1.5
						v3	87	56	1.55	48.6	855	593	1.4
m6	122	8352	150.6	12.8	11.77	v1	460	20	23	51.5	2386	1063	2.2
						v2	458	21	21.8	51.5	2405	1065	2.3
						v3	468	34	13.8	50.8	3454	1370	2.5
m7	389	56 K	477.0	81	5.89	v1	606	67	9.04	58.9	4727	2251	2.1
						v2	62	13	4.77	30.5	509	212	2.4
						v3	69	16	4.31	32.3	564	226	2.5
m8	901	106 K	1229	139	8.84	v1	555	140	3.96	32.5	93491	253	370
						v2	373	114	3.27	27.5	104925	223	471
						v3	359	111	3.23	20.0	104212	231	451

ferential algorithm [8], that stores differences between the good circuit and a faulty circuit at sequential elements. In the latch model, simulating a difference involves updating (and subsequent simulation) a single latch only. In the array model, any difference at a memory location triggers a resource-intensive evaluation of the entire array. Hence, some of the speed advantage of logic simulation is lost during fault simulation. It is not known if a different outcome would be observed with a concurrent fault simulator.

6 Conclusions

An algorithm for extracting array memory models using tree isomorphism was presented in this paper. Compared to other forms of isomorphism, tree isomorphism is linear in terms of the size of the graph. A general template for an array model has also been presented in this paper. Several memory circuits were modeled both as latches and arrays, and their behaviors were verified to be identical. A speed advantage of an order of magnitude was observed when logic simulation was performed. A somewhat lower, but substantial, speed advantage was observed for fault simulation also. This technique can be used to model any memory block whose contents are accessed at ports only, because port behavior is identical for the two models. More work is necessary to extend this algorithm to *content addressable memories*, and other similar designs, where port accessibility is not necessarily true.

References

[1] R. E. Bryant, "Algorithmic Aspects of Symbolic Switch Network Analysis," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, no. 4, pp. 618–633, July 1987.

[2] P. Agrawal, S. H. Robinson, and T. G. Szymanski, "Automatic Modeling of Switch-Level Networks Using Partial Orders," in *IEEE Transactions on Computer Aided Design*, July 1990, vol. 9, pp. 696–707.

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

[4] M. Pandey and R. E. Bryant, "Exploiting symmetry when verifying transistor-level circuits by symbolic trajectory evaluation," in *IEEE Transactions on Computer Aided Design*, July 1999, vol. 18, pp. 918–935.

[5] M. Velev and R. E. Bryant, "Efficient modeling of memory arrays in symbolic ternary simulation," in *First International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, April 1998.

[6] P. Wohl and J. Waicukauski, "Using Verilog Simulation Libraries for ATPG," in *Proc. International Test Conf.* 1999, pp. 1011–1020, IEEE Computer Society Press.

[7] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Design and Analysis of Computer Algorithms*, Addison Wesley Publishing Company, 1974.

[8] W. T. Cheng and M. L. Yu, "Differential Fault Simulation - A Fast Method Using Minimal Memory ," in *Proc. 26th Design Automation Conf.*, June 1989, pp. 424–428.