Implementation of Switching Circuit Models as Transfer Functions

David Kebo Houngninou
Department of Computer Science and Engineering, SMU
Dallas, Texas, USA
Email: dhoungninou@mail.smu.edu

Mitchell A Thornton
Darwin Deason Institute for Cyber Security, SMU
Dallas, Texas, USA
Email: mitch@lyle.smu.edu

***Abstract*—A transfer function is a mathematical function relating the output or response of a system to the input or stimulus. It is a concise mathematical model representing the input/output behavior of a system and is widely used in many areas of engineering including system theory and signal analysis. Binary Decision Diagrams (BDDs) are a canonical representation of Boolean functions. We implement a framework to build transfer function models of digital switching functions using BDDs and demonstrate their application on simulation and implication.**

I. Introduction

Our approach is motivated by the need to develop a truly unified EDA tool for mixed signal circuit design. Currently, industrial tools such as SPECTRE use two different internal engines; a SPICE-like engine and a Verilog-like engine. Our method will allow for all mixed signal circuit elements to be represented as transfer functions. The main contribution of this paper is threefold. First, we implement a new approach to represent switching circuit models as sparse matrices. Second, we create a representation for the Kronecker multiplication of Algebraic Decision Diagrams (ADDs). Third, we provide experimental results to showcase the advantages of decision diagrams over sparse matrices to represent transfer functions of switching circuit models. Previous work [6] described a new theory for representing switching functions with linear algebraic transfer functions. This paper describes how the theory can be used to build an efficient EDA tool for representing and manipulating switching functions as transfer functions.

Our chapter is organized as follows: Section 2 elaborates on the implementation steps including parsing, levelization and partitioning that are essential to generate the transfer function. Section 3 gives the basic properties of ADDs also referred as Multi-Terminal Binary Decision Diagrams (MTBDDs) and presents our implementation of the Kronecker multiplication of ADDs using radix polynomials. Finally, we present some experimental results to demonstrate the performance of decision diagrams on a set of benchmarks.

II. Constructing the Transfer Function

Building a transfer function consists of multiple steps starting from parsing a Verilog netlist. The main steps are: fanout detection, netlist levelization, netlist partitioning, crossover detection, translating nets to sparse matrices or binary decision diagrams and building partitions representing intermediate functions.

A. Fanout Detection

Typically, the output of a logic gate is connected to the input(s) of one or more logic gates. Fanout points are treated as network elements since these structures have differing numbers of outputs. To obtain the correct transfer function we must account for fanouts. The input Verilog netlists are in the form of a set of Boolean equations and these netlists do not explicitly define fanouts as is the case in other netlist languages like ISCAS85. The first step in the process is to identify all the fanouts in the netlist and rewrite the netlist to include those fanouts. This is done by parsing all the Boolean equations in the netlist and grouping all the gates that have identical nets in their input port list. For every set of duplicate nets found, we create a new fanout node with a unique identifier. The outputs of the fanouts have the same values as the input. Each output wire of a fanout is assigned with a distinct *ID* number.

B. Netlist Levelization

During event-driven simulation, gates are not always simulated in the order they are listed in a netlist. To simulate a circuit, we start by assigning binary values to the primary inputs and proceed by propagating those values until they reach the outputs. A single gate is not simulated until all of its input values are set. If the output of a gate named ‘A’ feeds another gate named ‘B’, then the output of gate ‘B’ depends on the output of gate ‘A’. To obtain the correct output value for gate ‘B’, the order of the simulation matters. In the case above, gate ‘A’ must be simulated prior to gate ‘B’. The process of ordering and determining the proper gate arrangement for simulation is called levelization. We start this process by assigning all primary inputs with an arbitrary value of zero. This value is called a level number. By propagating from the primary inputs towards the outputs, levelization assigns a level number to each gate and each wire encountered. This is done the same way in netlist simulation algorithms for EDA tools. To levelize a set of gates and nets in a circuit, four rules are applied:

1) A net or a wire can be assigned a level number only if its driving gate has been assigned a level number.
2) A gate can be assigned a level number only if all its inputs have been assigned level numbers.
3) The level number of a net or a wire is equal to the level number of its driving gate incremented by one.

4) The level number of a gate is the maximum of all its inputs' level numbers incremented by one. For instance, if a gate 'D' has two inputs 'B' and 'C' with level numbers 4 and 7 respectively, then the level number of gate 'D' is 8 ie. max(4,7) + 1.

To accomplish levelization, we use a recursive approach by applying the four properties above starting from the primary inputs. Once this process is completed, serial partitions are identified by finding cuts and grouping all nets that have identical levelization indices [6].

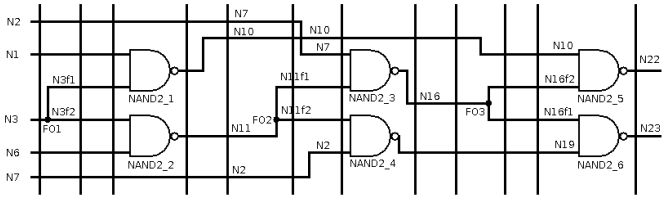

Fig. 1. Schematic of benchmark circuit c17.v with partitions cuts

C. Netlist Serial Partitioning

Nets and gates with the same level numbers are grouped in parallel stages called partitions. Partitioning separates the network into series or partitions of sub-circuits. A partition is made of the following types of elements: gates, fanins, fanouts and pass-through wires. All the primitive logic gates with the same level numbers are identified and grouped in identical partitions. Pass-through wires are wires that cross one or multiple partitions. Completing the serial partitioning requires two or more passes through the netlist and is thus of temporal complexity $O(n)$ where n is the number of nets. The spatial complexity is also $O(n)$ as the structural Verilog netlist is parsed into an internal graph memory structure where nodes represent gates, primary inputs and outputs. Graph edges correspond to the topological nets in the circuit.

D. Crossover Detection

Crossovers are the intersections of conducting wires. We can represent multiple crossovers as a series of single crossovers. Levelization does not detect crossovers, so we need an additional step to account for crossovers before the calculation of the overall transfer function. The intermediate permutation matrices for crossovers will be injected in between existing partition stages. In the case where there is no crossover, the permutation matrix is an identity matrix and the direct product does not modify the transfer function.

1) Crossover detection using linear equations: To identify crossovers between stages we use a set of linear equations (Figure 2). Lets consider two serial partitions: an origin partition and a destination partition. All nets in the origin partition must have a mapping in the destination partition (from outputs of stage m to inputs of stage $m + 1$). First, we assign an order to every nets in the origin partition, starting from the topmost element. Then, we assign an order to every nets in the destination partition, starting from the topmost element. The orders are used as y-coordinates in a two dimensional system. Using these coordinates, we compute a linear equation $y = a \cdot x + b$ for each pair of nets mapping from the origin to the destination partition. The equations are used to find the intersections of the lines. Each line intersection represents a crossover.

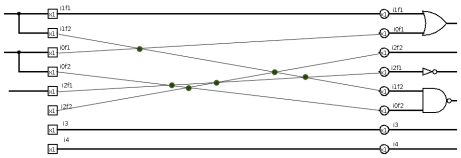

Fig. 2. Crossover detection using linear equations

2) Computation of the Permutation Matrices: Once we detect crossovers, we need to construct the corresponding permutations matrices. In order to get the correct transfer function, crossovers must be processed in the order they occur. Figure 3 shows the arrangement of the crossings. We process multi-wire crossovers one at a time.

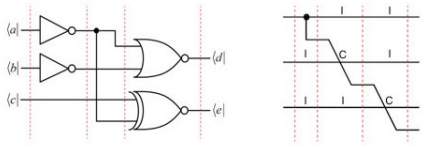

Fig. 3. Computation of a crossover matrix

Lines labeled 'I' represent wires and lines labeled 'C' represent crossovers. The transfer function for a wire is the identity matrix. The transfer function for a crossover is a predefined crossover matrix. In Figure 3, we have two crossovers. To compute the permutation matrix T we use the following equation:

$$T = (I \bigotimes C \bigotimes I) \cdot (I \bigotimes I \bigotimes C)$$

E. Building the Transfer Function for Partitions

The previous process of serial partitioning has grouped all nets and gates with the same level numbers in some parallel stages. Now that the partitions are formed, we need to compute their corresponding transfer function. To compute the transfer function for a partition, we perform the outer product of all the networks elements starting from the topmost element. Each partition transfer matrix requires p outer product operations where p is the number of parallel elements. To build the overall transfer function, we multiply all these partitions together in a specific order [6]. Starting from the leftmost partition, the first partition is multiplied by the next one to form an intermediate transfer function. The intermediate is then multiplied by the next partition in the row and so on. A transfer function requires m direct product operations where m is the number of partitions. When a crossover partition is encountered, it must also be inserted in the row and multiplied as well. When calculating the transfer function, we also take into account memory management. At every iteration, once an intermediate

function is calculated, we must discard the previous partition to free up unused memory. This technique is especially beneficial when dealing with large netlists.

III. Building partitions using Algebraic Decision Diagrams

Two approaches to efficiently represent switching functions are cube list representations and binary decision diagrams. Binary decision diagrams are widely used in logic synthesis and formal verification of integrated circuits. A BDD has a graph representation similar to a binary tree except that it is a directed acyclic graph [2]. It has one root, branch nodes and terminal nodes. The root node represents the Boolean function, the leaf nodes are either 0 or 1 and correspond to the constant Boolean functions. A BDD must obey two important rules. First, the diagram must be ordered: this means that variables must be encountered in the same order along the paths. Second, variables may occur at most once along a given path. Third, the diagram must be reduced: this means that all redundant nodes are removed and isomorphic subgraphs are shared to save space. Figure 4 illustrates the difference between a decision tree and a binary decision diagram.

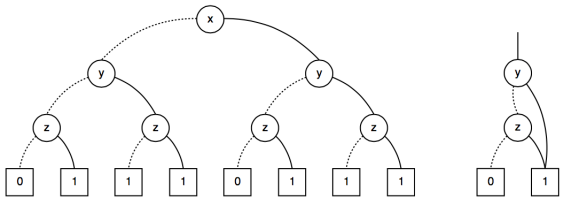

Fig. 4. A decision tree converted to a Binary Decision Diagram

Many tasks in synthesis, optimization, testing tools, design and verification of digital systems already manipulate large Boolean functions. However, to add value and improvements to future EDA tools, we need efficient ways of representing and manipulating such large functions. For this implementation of switching circuit models as transfer functions, we focus on the use of binary decision diagrams. Binary decision diagrams offer a canonical representation of Boolean functions and can be compressed by using the reduction and reordering rules. These attributes make BDDs suitable to save storage and improve efficiency when dealing with large formulas. Since the worst-case complexity is $O(2n)$ for switching functions, the motivation of this work is to take advantage of the reordering and reduction rules and provide a compact representation of functions. BDDs have multiple extensions. For this implementation, we use Algebraic Decision Diagrams (ADDs) also referred to as Multi-Terminal Binary Decision Diagrams [4]. As an experimental tools, we use CUDD: the Colorado University Decision Diagram Package written by Fabio Somenzi [5]. CUDD is a C/C++ library for creating different types of decision diagrams including: binary decision diagrams (BDD), zero-suppressed BDDs (ZDD) and algebraic decision diagrams (ADD).

1) The Kronecker Product: After applying serial partitioning to the netlist, we obtain partitions containing gates, fanins, fanouts and pass-through wires. To compute the transfer function for a partition, we use the Kronecker product also referred as outer product. The Kronecker representation is suitable for systems composed of parallel components, each component representing a matrix [3]. Since each parallel element in the partition can be interpreted as a matrix, we perform the outer product of all the matrices starting from the topmost element. In linear algebra, the outer product is the tensor product of two vectors. Therefore, the product $u \bigotimes v$ is equivalent to a matrix multiplication $u \cdot v^{\mathsf{T}}$. Starting from the matrix representing the topmost partition element, we multiply it by the transpose of the next matrix in the stage. Each operation is carried by pair, so the resulting matrix is then multiplied by the transpose of the next matrix and so on until we reach the bottommost element in the partition. The BDD representing a switching function is isomorphic to the matrix representation of the same function. Since our implementation focuses more on nodes reduction and reordering, we interpret each network element as BDDs rather than matrices.

2) The CUDD package: Using routines provided by CUDD, we build a library of BDDs corresponding to the most common netlist elements. We can create BDDs for primitive logic gates such as `AND`, `OR`, `XOR`, `NOT` using routines for conjunction, disjunction and complement. Algorithms of linear complexity are already available in CUDD and referred to as: `Cudd_bddAnd`, `Cudd_bddOr`, `Cudd_bddXor`, `Cudd_bddNot` and can be used to iteratively construct new BDDs from existing ones. The operation is performed by first creating a unique variable for each gate input, reference it, then applying the above routine to the inputs. The functions return a pointer to the resulting BDD if successful. These small BDDs are used as building blocks to compute larger partitions of parallel elements. Since we are now dealing with binary decision diagrams rather than matrices, we implemented a new method to perform the Kronecker product of BDDs. This multiplication is carried out by using radix polynomial.

3) Algebraic Decision Diagrams multiplication using radix polynomial: An algebraic decision diagram is a binary decision diagram whose terminal nodes can be arbitrary integer values instead of just 0 and 1 [1]. It is the perfect data structure to represent and manipulate large sparse matrices efficiently. Since it shares similar attributes with BDDs, we can easily convert one diagram type to another and vice versa. CUDD provides integer and floating point multiplication in algebraic decision diagrams. If f and g are two 0-1 ADDs, the function returns the inner product $f \cdot g$. This is done using a routine called `Cudd_AddTimes`. We modify the latter function to implement the Kronecker product. To get the outer product of two ADDs we use the following formula: $value = 2 \cdot cuddV(F) + cuddV(G)$.

'$cuddV(F)$' and '$cuddV(G)$' represent the two operands ADDs. '$value$' is a pointer to the resulting ADD and represents the result of the multiplication. Figure 5 illustrates the Kronecker product of an `OR` gate with an `AND` gate.

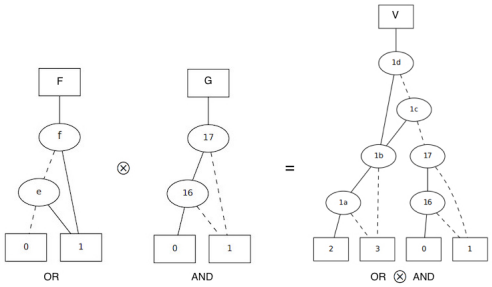

Fig. 5. Kronecker product or two ADDs

IV. Improvements to the CUDD package

In order to represent all network elements as binary decision diagrams, we added some additional functions to the CUDD library. One function represents a fanout and another function represents a crossover. These functions are essential for building the partitions. The fanout function takes an argument N, where N is the number of fanout wires and returns the corresponding BDD. The crossover function returns a BDD representing two crossing wires. Figure 6 shows both functions and their corresponding diagram.

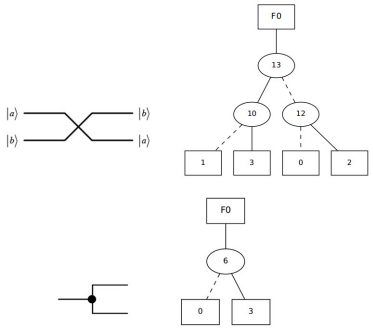

Fig. 6. BDD representation of a crossover and a fanout

V. Experimental results

To demonstrate our method, a set of benchmark circuits is used. The two-level benchmark netlists are converted into multi-level combinational logic circuits in the form of a Verilog structural netlist before the technique is applied to them. This initial conversion process is accomplished by converting the native .pla files into corresponding Verilog files using Synopsys Design Compiler. The converted files are then in the form of a set of two-level Boolean equations expressed in Verilog syntax. The multilevel netlists are saved as structural Verilog descriptions and used as input to a parser that computes the corresponding binary decision diagrams. For this experiment we use two algorithms for building the BDDs. The CUDD package offers multiple dynamic reordering algorithms. BDDs and ADDs, which share the same unique table are simultaneously reordered for efficiency. These algorithms iteratively improve variable orders to avoid the BDDs size to grow out of boundaries during computation. The first algorithm uses variable reordering and the second one does not use variable reordering. The table below summarizes timing data, the total number of nodes and memory usage for the use of sifting variable reordering.

TABLE I
BDDs with sifting variable reordering

Benchmark	Inputs/ Outputs	# of partitions	Total # of nodes	Memory (MB)	Time to build partitions (ms)	Time to build BDDs (ms)
i3.v	2/3	6	99	8.97	0.35	0.21
xor5.v	5/1	6	168	8.97	0.42	0.24
c17.v	5/2	12	595	9.06	1.23	0.68
majority.v	5/1	12	1520	9.11	1.26	1.05
test1.v	3/3	16	651	9.08	1.53	1.03
rd53.v	5/3	18	5137	9.33	2.45	2.87
con1.v	7/2	14	32841	10.85	3.17	139.10
radd.v	8/5	28	185640	16.05	7.56	299.05
rd73.v	7/3	24	16129	10.15	4.96	37.37
cm163a.v	16/5	26	610340	59.19	11.68	11208.75
cm162a.v	14/5	28	151678	63.27	27.56	27013.89
mux.v	21/1	26	622061	278.52	23.78	291722.15
cm85a.v	11/3	26	102739	77.80	12.85	63527.78
x2.v	10/7	23	221596	125.04	24.75	121033.49

VI. Conclusion

We have described how the theory in [6] can be used to efficiently build and manipulate switching circuits using BDDs. Our approach consists of representing a digital logic network as a transfer function using a sparse matrix or a binary decision diagram. To obtain an output response for a simulation, we represent the input stimulus as a vector, and the transfer function can linearly transform that input vector into an output vector. The advantage of computing transfer functions is that we can use the same function framework for both digital network simulation and the reverse process: implication. Future work will focus on computing the resulting transfer function as a binary decision diagram by merging all the BDD partitions obtained from this experiment either by multiplication or by composition. The attributes of these diagrams are: the ability to have a more compressed representation of the transfer function especially when dealing with large formulas, the ability to use multiple variable reordering algorithms and the benefits of a smaller memory footprint.

References

[1] R Iris Bahar, Erica A Frohm, Charles M Gaona, Gary D Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebric decision diagrams and their applications. *Formal methods in system design*, 10(2-3):171–206, 1997.

[2] Edmund M Clarke, Masahiro Fujita, and Xudong Zhao. Multi-terminal binary decision diagrams and hybrid decision diagrams. In *Representations of discrete functions*, pages 93–108. Springer, 1996.

[3] Luca De Alfaro, Marta Kwiatkowska, Gethin Norman, David Parker, and Roberto Segala. *Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation*. Springer, 2000.

[4] Masahiro Fujita, Patrick C. McGeer, and JC-Y Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal methods in system design*, 10(2-3):149–169, 1997.

[5] Fabio Somenzi. Binary decision diagrams. In *Calculational System Design, volume 173 of NATO Science Series F: Computer and Systems Sciences*, pages 303–366. IOS Press, 1999.

[6] Mitchell Thornton. Simulation and implication using a transfer function model for switching logic. *IEEE Transactions on Computers*, PP, February 2015.