Binary Decision Diagrams – Review and Applications in VLSI Design

David Artz

ABSTRACT

In a 2008 lecture Dr. Donald Knuth called Binary Decision Diagrams (BDD) "one of the only really fundamental data structures that came out in the last twenty-five years". He went on to indicate that CMU's Dr. Randal Bryant's seminal paper (1986) had been the most cited paper in computer science. While BDDs were not new,[2,3] Bryant's contributions were threefold, (1) reduction approach to the BDD along with (2) fixed variable ordering (3) which allowed for efficient data structure and associated algorithms that have been instrumental in propelling solutions to many complex problems heretofore untouched. This is especially true in VLSI design, where this simple and elegant data structure has afforded us with solutions in synthesis, verification, DFT, etc. While ROBDD's (Reduced Ordered Binary Decision Diagrams) have led way to more recent advances (or more appropriately said, renewed interest) in other approaches (for example, And Invert Graphs used in conjunction with much more efficient Boolean satisfiability (SAT) solvers) BDD's are still important and given the plethora of robust open source libraries for most languages (C++, Perl, Python, Tcl, Java, etc.) I assert still an import tool to add to your repertoire.

Keywords

Fun, Interesting, Neat, Cool, BDD, ROBDD, Simple, Elegant, Powerful.

1. INTRODUCTION

If you're an engineer who has spent much of their focus closer to circuits and device physics, or have been in the industry for several decades, you may know little about the Reduced Ordered Binary Decision Diagrams as had I. A few years ago, when faced with the problem of characterizing a library (we were small project, no funding for library characterization software) I happened across the ROBDD (often referred to as just BDD) data structure and associated algorithms. This was a key piece of technology needed to easily automate the stimulus generation and side pin sensitization. A number of robust BDD open source libraries were available and after picking one pretty much at random, had a working solution completed in no time, all with little knowledge of the elegance, simplicity, and power that the underlying BDD's afford.

Recently I had an opportunity (due to some of the unique design challenges in the design of FPGA's) to delve into ROBDD's in more depth to solve a problem that did not lend itself readily to commercial EDA/CAD. That was the verification of conditional mode expressions for the look up table functionality.

I hope in some way my enthusiasm for the importance and wide applicability of BDD's to many problems we have in VLSI design will encourage the reader to incorporate this capability, much like ones understanding of linked lists, hash tables, or other data structures we use from time to time in their arsenal of problem solvers. Or, at least give one deeper appreciations of why some tools behave in the manor they do.

2. BACKGROUND

There are many forms we are used to seeing Boolean relationships in. Most of us are of course familiar with schematics, Boolean expressions, truth tables, Karnaugh maps, etc. All these representations are different forms for a proposition or logic statements (most are ill-suited for manipulation in the computer, e.g., a truth-table for an n input Boolean function will have 2^n rows). And by proposition statements I am speaking from Mathematical Logic, e.g., Proposition logic (a system based on propositions or statements which are declarative sentences that are either true or false). There are more advanced forms of mathematical logic, modal, equational, 1st order, and higher order logic and calculus. Why do I bother to mention this? If you go out to explore the literature on BDDs, you will see many times the author will write entirely in the language of propositional logic, or higher order logics. Some of the basic Boolean theory was of course established long ago[5] and sometimes this older nomenclature and terminology has carried forward. Indeed, the power of BDD's is that it's quite easy to utilize the long lineage of work and their formalisms, axioms, relationships, etc. in the context of this elegant data structure.

3. Boolean Representation

So what is A BDD? Let's answer that in context of starting from the most ubiquitous description of Boolean functions, the truth table in Figure 1. For example, the table below represents the expression $f = (a \wedge \neg c) \vee (c \wedge b)$.

> Note the expression is in sum-of-product form or disjunction normal form (DNF), i.e., a disjunction of conjunctive clauses where a clause is one or more literals and a literal is either a variable or negated variable. The Product-of-sums or conjunctive normal form (CNF) is similarly defined, but with the connectives switched.

To the right of the truth table is the same function represented as a binary decision tree.

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

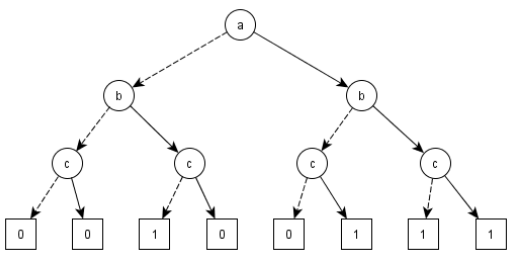

Figure 1

Some things to note in the graph, the vertices represent the variables in the expression. If you want to solve the function for various

assignments of 1 or 0 to the variables, you would start at the root node, in this case variable *a*, and depending on the variable assignment traverse the dotted edge for a zero assignment and the solid edge for a one assignment. So, in the case of *f(a=1, b=1, c=0)*, we would travel the solid edge from the root node, down to the variable *b*, at which point we would again travel the solid edge to variable *c*, and finally with the *c* being assigned zero, we would take the dotted edge terminating at a leaf node which has a value of one. Examining the truth table we can see that *f* = 1 for the given variable assignment. Note that this graph is ordered. That is the variables, *a*, *b*, and *c* all appear at the same depth from the root node, the common notation for this is *a* < *b* < *c*.

3.1 If Then Else

Let's look at this tree (Figure 1) with a little more scrutiny. Notice if I wanted to find all the variable assignments that satisfied the equation (i.e., *f* = 1) then I need only walk from every 1 leaf node back up the tree to the root node. The edges telling me what the variable assignment needs to be. Also note how the right hand side of this graph corresponds to the lower half of the truth table (the 1 branch from the root node). Conversely, the 0 branch from the root node *a* corresponds to the upper half of the truth table. It's as if the *a* variable is a select on a mux which has two inputs from two different functions represented by the subtrees off of *a*. Indeed, we can think of the nodes in this tree as just that, little mux2's controlled by the variable of that node. I can redraw the tree as the following schematic:

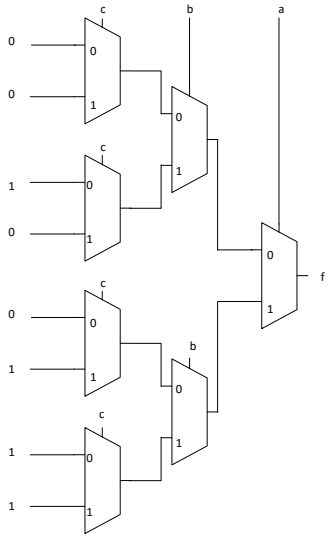

Figure 2

Of course, you already knew this if your familiar with the inner workings of FPGA's - that is, any function can be implemented by muxes (LUT). We may also refer to this as an ITE implementation (If Then Else). But, back to the subtree expressions, we can redefine the function *f(a,b,c)* as follows: $a \wedge f(1,b,c) \vee \neg a \wedge f(0,b,c)$. As should be apparent, this can be applied recursively through the entire depth of the tree.

3.2 Shannon Decomposition

This is of course what Boole and Shannon realized and is referred to as Shannon's Expansion Theorem and the more general form is $f = x_i \wedge f(x_i{=}1) \vee \neg x_i \wedge f(x_i{=}0)$. I like to think of this analogously as a way it may be handled in the real number domain, that is, representing a complex function in simpler form much the way we do with Tayler series expansion of e^x being $e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$. As we shall see, there are many analogs in the Boolean domain that are powerful operators which are very useful when combined with BDD's. The subexpressions in the example, i.e., $f(x_i{=}1)$ and $f(x_i{=}0)$ are called the positive and negative cofactor of *f*. The positive cofactor $f(x_i{=}1)$ is obtained by substituting (restricting or constraining) 1 in for the variable (or variables) of interest. In our small example the positive cofactor would be $\neg c \vee (c \wedge b)$. For the negative cofactor we substitute a 0, resulting in $(c \wedge b)$. There are several very useful properties of cofactors. For example, cofactor of NOT is the NOT of cofactors, the cofactor of AND is the AND of cofactors, etc., etc., etc. valid for all Boolean operators.

3.3 Characterization and DFT

Remember the little problem I mentioned, sensitizing the secondary pins of a hundreds of library cells to perform characterization? How would one formulate a solution to that given what we have so far talked about? Let's continue with our little function shown in symbol form below.

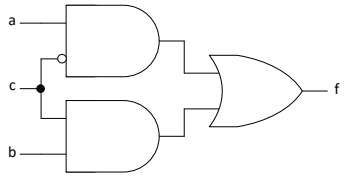

Figure 3

Given the above, how might we determine the voltages on any two of its inputs to perform a transient from the remaining third pin to the output? Think of the following approach to finding a propagation of *a* to *f* enabling assignment for secondary pins *b* and *c*.

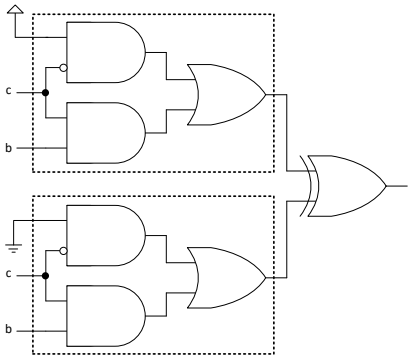

Figure 4

In the figure above there are two instances of the example function, the top instance has it's *a* pin tied high. The bottom instance has the *a* pin tied to low. Their outputs are connected to an XOR gate and thus the result will only be high when the two instances differ in value. Whenever this condition occurs (when output of XOR is high) we know the *b* and *c* states are such that a transition on the *a* pin would result in a corresponding output change. Thus, we have a way of identifying one or more states for the secondary pins which would allow for a transition from *a* to *f*.

If you're into DFT, you may have noticed this is the exact same approach we would take if we were looking for stuck at faults. The

goal would be to sensitize a path being tested such that we can verify its functionality.

The careful reader will note that these two instances are nothing more than positive and negative cofactors of the Boolean expression with respect to *a* XOR'ed together. And, if we were to create a binary decision tree for that expression, we could easily find valid constraints for pins *b* and *c* by walking from any of the 1 leaf nodes up through the tree and looking at the variable assignments along the way. Of course, the careful reader has also probably noted that our binary decision tree is just as large as a truth table would be. This then brings us to the Reduced and Ordered Binary Decision Tree.

4. ROBDD's

Before we review the basics of ROBDD's let's brush up on some logic nomenclature and symbolisms.

4.1 Speaking Logic

As mentioned, some of the published literature on this topic can be somewhat obtuse for the reader who is not well versed in propositional logic or predicate calculus, etc. Even an electrical engineer and UC Berkeley professor can be confused over correct explanation of the terminology and surprisingly there is quite a lot of aberrant usage among logic practitioners themselves [1].

We are all familiar with basic logic functionality and the logic expressions referenced so far have already used fairly common symbols for negation (NOT, $\neg$), conjunction (AND, $\wedge$), and disjunction (OR, $\vee$). There is of course exclusive or (XOR, $\oplus$) and perhaps the less familiar implication (if … then, $\rightarrow$), and bi-conditional (if and only if, $\leftrightarrow$, also called bi-implication). These are operators which function on literals (variables), formulas or some combination of both. A tautology is an expression that is always true (e.g., $x \vee \neg x$). A contradiction is simply not a tautology (i.e., $\neg(x \vee \neg x)$) (pun intended ☺). Satisfiability (as we alluded to already) implies a Boolean expression has at least one combination of variable assignments that would lead to 1. We have already seen a case where we were interested in satisfiability of a Boolean expression. That was the condition we were seeking for the example from Figure 4.

4.1.1 Quantification

Perhaps even more esoteric Boolean terminology, but very germane to BDD's are quantifiers, such as universal quantification (for all) $\forall xf$, sometimes referred to as consensus of f w.r.t. x, and existential quantification (there exists) $\exists xf$, alternatively known as smoothing of f w.r.t. x. Universal quantification and existential quantification can be thought of as operators on functions much like the integral symbol from calculus $\int$. And why I've mentioned calculus; you may be surprised to know we have already shown the usefulness of a Boolean differential (also known as unique quantification $\frac{\partial f}{\partial x}$). Just as a differential equation shows how the function changes with respect to the derivative of the variable in question, analogous in the Boolean realm is unique quantification, where we are looking to see if the Boolean is sensitive to changes in that variable. This of course was our example of finding the side pin constraints of a circuit to be characterized for a timing arc of interest. We used the Boolean difference to see if the two copies of our function was sensitive to differences in said variable and the found the values of the other variables under those conditions (satisfiable).

Quantification operators are all related to Shannon expansion or decomposition. These operators are powerful and can be used in derivation of BDD's themselves or in answering a whole host of very important questions about one or more BDD's (e.g., equivalency). In summary they are:

Universal	$\forall x_i f = f(x_i{=}1) \vee f(x_i{=}0)$
Existential	$\exists x_i f = f(x_i{=}1) \wedge f(x_i{=}0)$
Unique	$\frac{\partial f}{\partial x_i} = f(x_i{=}1) \oplus f(x_i{=}0)$

Note that quantification operators are easily applied too two or more values in the following way, e.g., with two variables x and y the existential quantification would be

$$\exists x, y f = f(x{=}0,y{=}0) \wedge f(x{=}0,y{=}1) \wedge f(x{=}1,y{=}0) \wedge f(x{=}1,y{=}1)$$

Adding to the confusion is that some of the logic symbolism is not consistently used, e.g., we all can discern $\top$, $\bot$ meaning true (T, 1) or false (F, 0) respectively and negation maybe appearing as "!", "~", or the horizontal line over a literal or entire formula, while conjunction may be indicated by "&", ".", or "*" and disjunction by "|" or "+". Worse still is inconsistent usage of implications, seeing it referred to with any of following symbols $\Rightarrow$, $\rightarrow$, $\therefore$, or $\supset$. Of course, as you have seen, there are synonyms for much of the terminology adding even more confusion for the novice. Please don't let this deter you from studying this technology; it's worthwhile to overcome these stumbling blocks!

4.2 Reduction

While we have seen the BDD's encode the relationships between the states of a variable and the resulting function they make up. We have also seen that they are no better at storing and retrieving that information then $O(2^n)$. Let's show the tree again from Figure 1 to see if we can do better.

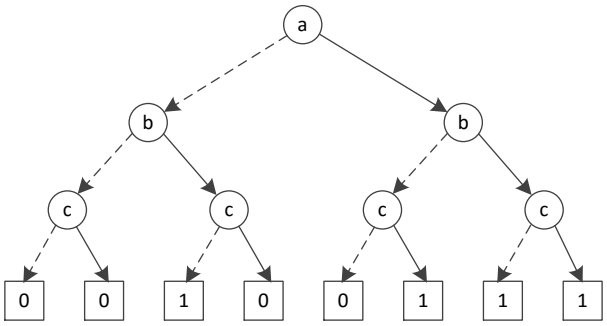

Figure 5

Clearly, there is redundant information in our graph. Look at the terminal nodes (square 1 or 0 leaf vertices). These could be reduced to just two nodes representing 1, and 0. Also, notice the left and rights sides of the tree, where the *c* variables are not needed (i.e., ITE(c,0,0) = 0, and ITE(c,1,1) = 1). Could this tree not be reduced to that of the graph in Figure 6?

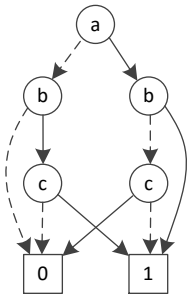

Figure 6

To summarize, an ROBDD is nothing more than a binary decision tree (which is a representation of Shannon expansion) that has had any common (isomorphic) subtree's merged (in this example we only had common leaf nodes which were merged). In addition, any node with identical children is removed (as was the case for the nodes ITE(*c*,0,0) and ITE(*c*,1,1). And, to reiterate, vertex ordering must be the same from the rooted node, no matter what path you traverse to a leaf node. This reason will be explained shortly. Graphically put we have, eliminated redundant tests, e.g.:

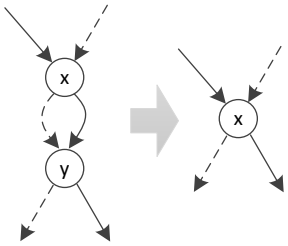

Figure 7 (eliminate redundant tests)

Merged equivalent leaves:

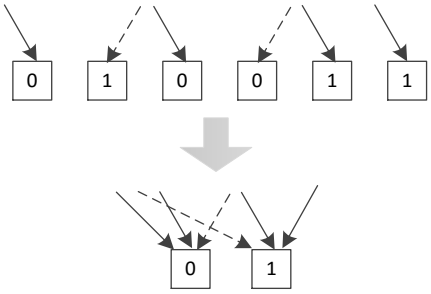

Figure 8 (merge equivalent leaves)

And finally merged isomorphic subgraphs (with some additional annotation to highlight a little rigorousness and not just intuition).

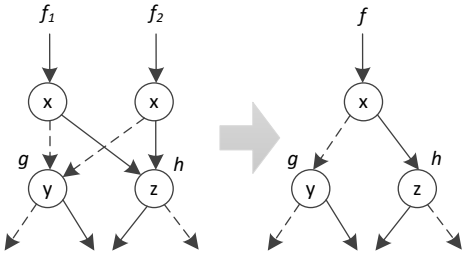

Figure 9 (merge isomorphic subgraphs)

$$f_1 = f_{\neg x} \wedge g(y) \vee f_x \wedge h(z) = f_2 \equiv f$$

These are reductions to a simpler graph are the contributions of Bryant. More importantly, he showed that if you maintain a variable ordering you will end up with a unique signature if you will, of your Boolean expression after reduction. So, in our three input pin example, if we were to compute the ROBDD graph for all possible three input Boolean expressions we would see a unique graph for each and every one. This indicates that we can compare these graphs against each other for comparison of functionality (an implicit formal verification). Indeed, coupled with Bryant's elegant and simple data structure for ROBDD's, this operation (a formal comparison) can be achieved in O(1) time! Of course, building the ROBDD's takes longer. Unfortunately, there are a class of expressions which has been shown cannot be done any faster than exponentially (e.g., multipliers). But, the wide adoption of ROBDD's and their various incarnations should be a clear indication that most real world problems exhibit much better runtime and memory consumption behavior.

4.3 Variable Ordering

You might be curious if the ROBDD graph for our example expression $(a \wedge \neg c) \vee (c \wedge b)$ can be made any smaller. Certainly it can. The choice of our variable ordering significantly impacts the resulting quality (as measured by node and/or edge count) of the ROBDD. We would generate the following ROBDD if we used the variable ordering $b < c < a$:

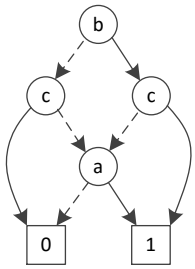

Figure 10 ($b < c < a$)

What if we wanted to write out the Boolean expression from the BDD, how would we proceed? Simple, traverse all the paths that lead to the functions 1 value. If we traverse a false path from a literal we write down its negated form otherwise the non-negated form is used and connected into conjunctive clauses for each path. These are finally put together with disjunction connectives. From figure 10 above, the resulting equation would be $(\neg b \wedge \neg c \wedge a) \vee (b \wedge \neg c \wedge a) \vee (b \wedge c)$. This of course is the SOP form, for the POS for we would reverse the procedure, tracing each path to the false leaf node and connecting with the disjunction operator to form clauses, then conjoining the clauses with conjunctions.

But wait, we can still do better, using the variable ordering $c < a < b$ we obtain the graph in figure 11, which results in our original expression $(\neg c \wedge a) \vee (c \wedge \text{b})$. Please note, this is not an approach for creating minimum CNF or DNF expressions.

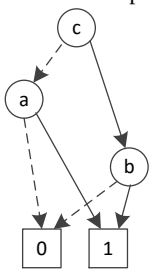

Figure 11 ($c < a < b$)

This has been a whole area of fruitful research culminating with many static and dynamic approaches to finding the optimal variable order. By static I refer to the initial order, perhaps gleaned from the schematic circuit itself (in the case your deriving ROBDD's from a design schematic), or dynamically, either during the building or modification of the ROBDD or as a separate step entirely. In any case, the correct variable ordering can mean the difference between traversing a graph an exponential number of times to obtain some answer, or a linear number of times. The difference can be very stark on real world examples as shown by a small four input mux below with poor variable ordering.

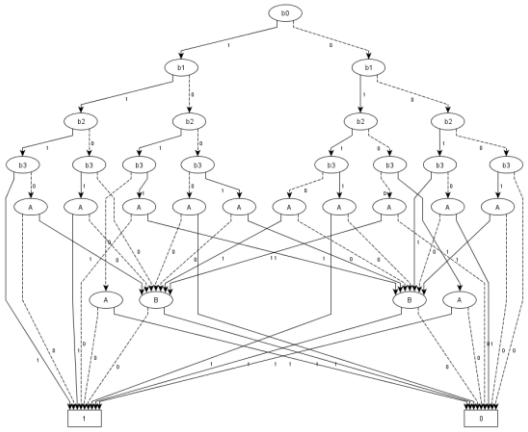

Figure 12 (poor variable ordering on mux4)

Versus an optimal variable ordering for the same four input mux (see Figure 13 below). Although, sometimes a non-optimal variable ordering may be advantageous, e.g., if we treat this small mux4 example as a simplified version of the input stage of the la_l_lut molecule we can see that if b_n bits representing the CRAM pattern, when ordered prior to the selects *A*, and *B*, it's more apparent that there are certain CRAM bit values where the selects play no role in the overall function. For example, traversing the extreme left or right paths from the rooted node b_0 we would generate the constants 1 or 0, independent of the selects *A* and *B* values.

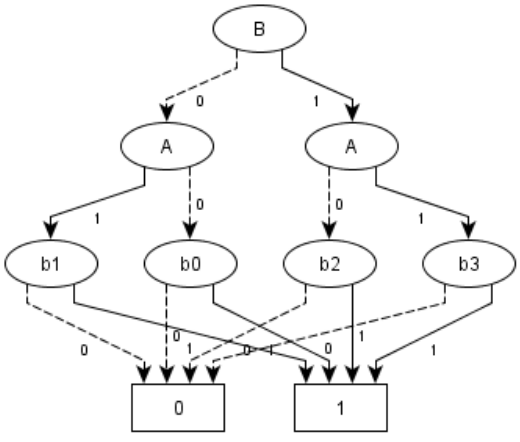

Figure 13 (optimal variable ordering on mux4)

The reader should not be put off using ROBDD's over these details of building and optimizing the graph. Most open source BDD packages take care of these details for the user. Detailed knowledge about the underlying technology is reviewed for those occasions when a user encounters issues that may require debugging or as I said at the outset, understanding some tools behavior's (e.g., Synopsys esp-cv, and why it can easily run out of memory or take forever to complete on certain schematics without intervention).

4.4 Formal Equivalency

While I've already mentioned that the ROBDD serves as a signature if you will, that can be an implicit check of equivalence between circuits because of the ROBDD's canonical form, what would the results be if we explicitly verified two identical Booleans. Using our example again, let's say we wanted to compare against a different implementation, but same function:

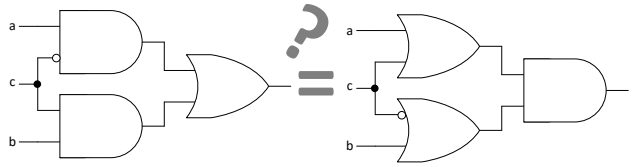

Figure 14

If you build a ROBDD for both of these circuits you will (given the variable ordering c < a < b) expect to get identical ROBDD's from Figure 11. But, what would happen if you explicitly did a formal comparison and we used the approach similar as with the challenge of finding a propagation enabling assignment?

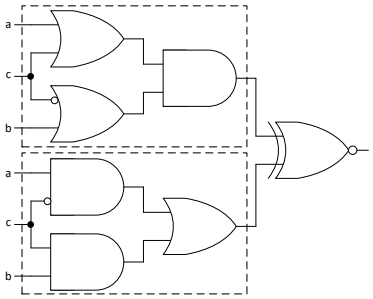

Figure 15 (explicit formal verification)

In this case we tie each of the circuits together with a XNOR which will result in a 1 only when the inputs are identical. If you build the ROBDD for this complete circuit, you will see that the resulting ROBDD is a very uninteresting graph but is indeed a tautology (true for every input combination) and thus formally confirms equivalency.

Figure 16 (tautology ROBDD)

As ROBDD popularity spread like wild fire in the early 1990's a plethora of tweaks and enhancements and modifications were made to address its short comings or extend it for certain domains. Stochastic optimization to variable ordering, further reduction of the data structure (e.g., do we need to explicitly indicate the false paths). One could imagine all the permutations you could do by allowing the edges to take on different values, relaxing the ordering of variables within the BDD, different decomposition approaches, etc., etc., etc. Resulting in more derivatives or flavors of BDD's then time permits to discuss in this paper, e.g., ZBDD's, EVBDD's, FEVBDD's, MTBDD's, Free BDD's, etc., etc., etc. Improvements have also been made to the fundamental data structure which very economically stores the Boolean; resulting in support for extremely large BDD's so large where the structure is stored on disk and not main memory. Improvements to the garbage collection cache

handling. Parallelization and thread safe enhancements, again the list goes on seemingly ad-infinitum.

Just as the list of extensions to BDD's has grown prolifically, so has the application to real world problems in VLSI design. While we have only scratched the surface in terms seeing a few small examples the list obviously contains Synthesis, e.g., in the technology mapping phase, as it was seen that the BDD is decomposition into simpler expressions, and though ordering and reduction we can have canonical representations for Boolean functions. This approach is one way to map a BDD into a library of gates. Obviously, verification is readily tackled via BDD's as we have seen. Not just for cones of combinatorial logic, but also finite state machines. In the case of FSM we essentially encode the variables just as we did with our examples, but also introduce the idea of a next state for the variables, e.g., variable v, would have v' representing its next state, say in the case of a counter.

Perhaps I'll leave you with a more concrete example of how easy BDD's can be used to represent something other then what we have discussed so far. How would you represent a function where we have values other than just true or false, perhaps you need five values, the $\top, \bot$ (true and false) but also z (for high impedance) and x (unknown) and let's say x' (unknown next state). We could encode these multiple values as binary variables as show below.

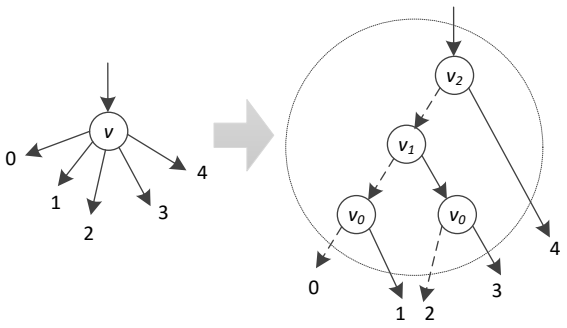

Figure 17

Now it should be apparent that the BDD can represent the function of many circuits, used in verifying many characteristics, simulation, verification, etc. and all this with the same data structure and underlying supporting algorithms simplifying ones coding challenges.

5. BDD Libraries

Hopefully you may be encouraged to try some small problems yourself, but daunted at the task of implementing the basic BDD capabilities. I would be also, it typically takes a very experienced software developer and advanced knowledge of BDD's and ROBDD's in general many months to implement a base library in support of same. But, given the popularity of BDD's, there's an abundance of good implementations readily available in your favorite language. While most have been implemented in C/C++, their API's have been ported to Perl, python, tcl, Java, etc.. Your problem is now one of selecting the best package for your needs. Dr Geert Janssen does a great job of comparing most of the available software for BDD creation, manipulation, and support. In all, he reviews 13 packages and in summary concludes only a few are worthwhile of industrial level problems. I personally have had great success with both CUDD and BuDDY, both API's are similar enough that if you learn one, the other is quite easy to migrate to. CUDD affords more functionality then BuDDY, indeed, more than any other available package. And both are blazingly fast. CUDD also has more recent support and maintenance, and more comprehensive documentation.

6. CONCLUSIONS AND FUTURE WORK

We have looked at BDD's, and in particular the ROBDD, introduced some of the Predicate or 1st Order Logic terms and vernacular necessary to further probe the extent of BDD's usage in VLSI design. A few of examples (verification, test, equivalency) were presented to hopefully wet ones appetite over further research into BDD's as they had mine. While only touching on a few examples, rest assured BDD's have found their usefulness in other VLSI problems, e.g., the optimal synthesis of pass gate transistor circuits and their layout, signal probability determination (as in Ansys Red Hawk's probability propagation switching activity mode), even RLC interconnect parasitic reduction.

7. REFERENCES

Predicate Logic

[1] Eric C. R. Hehner, "Boolean Formalism and Explanations", 1996 Proceedings of the 5th International Conference on Algebraic Methodology and Software Technology, 351-374.

Historical

[2] C. Y. Lee, "Representation of Switching Circuits by Binary Decision Diagrams," Bell Syst. Tech J., vol. 38, pp. 985-999, July 1959.

[3] S. Akers, "Binary Decision Diagrams," IEEE Trans. Computers, vol. C-27, no. 6, pp. 509-516, June 1978.

[4] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," IEEE Trans. Computers, vol. C-35, no. 8, pp. 677-691, August 1986

[5] George Boole, "An Investigation of the Laws of Thought: On which are Founded the Mathematical Theories of Logic and Probabilities", 1854

Variations on Binary Decision Diagrams

[6] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. "Algebraic decision diagrams and their applications". In Proc. IEEE/ACM International Conference on CAD, pages 188{191. IEEE Computer Society Press, 1993.

[7] Alan Mishchenko, "An Introduction to Zero-Suppressed Binary Decision Diagrams", Technical report, Portland State University, June 2001

[8] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. "Algebraic decision diagrams and their applications. Formal Methods in System Design", 171-206, 1997.

[9] A.Narayan, "Recent Advances in BDD Based Representations for Boolean Functions: A Survey", in Proc. 12th International Conference on VLSI Design, Goa, India, 1999, pp. 408-413.

Implementations

[10] Geert Janssen, "A Consumer Report on BDD Packages". Proceedings of the 16th Symposium on Integrated Circuits and Systems Design (SBCCI03). Sao Paulo, Brazil, 2003

[11] Lovato, A., Macedonio, D., Spoto, F., "A Thread Sage Library for Binary Decision Diagrams", Giannakopoulou, D.,

Salaun, G. (eda) SEFM 2014. LNCS, vol. 8702, pp. 35-49. Springer, Heidelberg (2014)

[12] “Parallel Disk-Based Computation for Large, Monolithic Binary Decision Diagrams” D. Kunkle, V. Slavici and G. Cooperman. International Workshop on Parallel Symbolic Computation (PASCO '10) Grenoble, France, 2010.

[13] Haim Cohen, “The BuDDy Library & Boolean Expressions”, Dr. Dobb's Journal, September 2004

Design for Test and Testability

[14] Juan A. Carrasco and Víctor Suñé, “An ROBDD-Based Combinatorial Method for the Evaluation of Yield of Defect-Tolerant Systems-on-Chip”, IEEE Transactions on VLSI Systems, Vol 17, No. 2, February 2009.

[15] Janusz Sosnowski, Tomasz Wabia, Tomasz Bech, “Path Delay Fault Testability Analysis”, DFT 2000: 338-346

Activity Factor or Signal Switching Probability

[16] Christine H. Tran, “Incremental Switching Factor Calculation for Power Estimation”

[17] Felipe Machado, Yago Torroja, Teresa Riesgo, José Gutiérrez Abascal, “A BDD Proposal for Probabilistic Switching Activity Estimation”

[18] Yingtao Jiang, Yuke Wang, Xiaoyu Song, Y. Savaria, “Computation of Signal Output Probability for Boolean Functions Represented by OBDD”

Pass Transistor Circuits

[19] Rupesh S. Shelar and Sachin S. Sapatnekar, “BDD Decomposition for Delay Oriented Pass Transistor Logic Synthesis”, IEEE Transactions on VLSI Systems, 2005

[20] D. Markovic´, B. Nikolic´, V.G. Oklobdz˘ija, “A general method in synthesis of pass-transistor circuits”

[21] P.W.C. Prasad, M. Raseen, S. Sasikumaran, “Delay Minimization In Pass Transistor Logic Use Of Binary Decision Diagram”

[22] Christoph Scholl Bernd Becker, “On the Generation of Multiplexer Circuits for Pass Transistor Logic”

Variable Ordering

[23] Michael Rice and Sanjay Kulhari, “A Survey of Static Variable Ordering Heuristics for Efficient BDD/MDD Construction”

[24] Lei Zhang, Zhenghui Lin. Zongwei Lv, “How to Group Variables for Reducing BDD”

[25] Richard Rudell, “Dynamic Variable Ordering for Ordered Binary Decision Diagrams”

[26] Pi-Yu Chung, Ibrahim N. Hajj, and Janak H. Patel, “Efficient Variable Ordering Heuristics for Shared ROBDD”

[27] Masahiro Fujita, Yusuke Matsunaga, and Taeko Kakuda, “On Variable Ordering of Binary Decision Diagrams for the Application of Multi-level Logic Synthesis”

Finite State Machine, Reachability, etc.

[28] Wilsin Gosti, Tiziano Villa, Alex Saldan, Alberto L. Sangiovanni-Vincentelli, FSM Encoding For BDD Representations