Binary Tree Structure for Formal Verification of Combinational ICs

FATMA A. EL-LICY, and HODA S. ABDEL-ATY-ZOHDY
Microelectronics System Design Laboratory
Department of Electrical and Systems Engineering
Oakland University, Rochester, MI 48309-4401
USA
E-mail: zohdyhsa@oakland.edu

Abstract:- Binary Tree structure is used to represent circuit design specifications as well as implementations and has been manipulated with different tools to achieve design verification.
The verification methodology is based on design specifications and layout descriptions and provide two binary trees of the design specification-implementation functionality. Desired design behavior is represented as propositional logic in Disjunctive Normal Form (DNF), which has been chosen to accelerate and facilitate the matching process for design correctness.

Key-Words:- Binary tree structure, formal verification, digital integrated circuits.

1 Introduction

The production of complex, Very Large Scale Integrated (VLSI) chips is very expensive and time consuming. It requires multistage hierarchical design methodology, as well as precise models for each stage. It is, therefore, important to detect and eliminate design errors prior to production. While extensive simulation may provide a degree of confidence about the correctness of a design, it is not inclusive. Thus, formal hardware verification methods are needed. Formal verification is a method by which the implementation of a design is formally proved to satisfy the design specification.

Verification is also complementary to automated synthesis, where, an implementation of a design is produced and is assumed correct by construction. For high quality designs, however, custom (manual) design methods are preferable. Formal verification can then be used to confirm the satisfiability of these designs.

In this paper, a new approach for combinational circuit verification is presented. The methodology, as well as the structure of electrical integrated circuit design specification/implementation constitutes manipulation of logic binary trees to accelerate and facilitate verification process.

Binary tree structure, as circuit representation, has been adopted in some verification systems as a prestructure [1,2,3]. This structure, however, is then transformed into different graph structures e.g. Binary Decision Digram, (BDD)

or Ordered Binary Decision Digram, (OBDD) [1,2]. Such representation is not efficient for serial or computational circuits, because of the explosion of the size of transition relations, which is exponential with respect to the number of primary inputs. Formal verification has to be performed in a formal environment, Figure 1 present a block diagram of the system. The main components are:

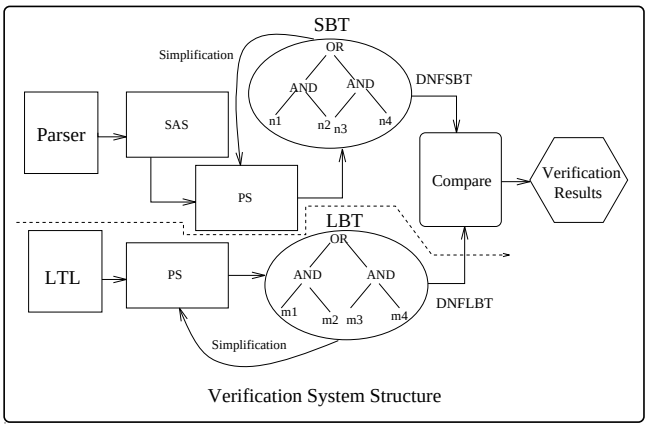

Figure 1: Block diagram for the verification system structure

1. Parser [4]: The Parser is applied to circuit design specification, which could be a Hardware Description Language (HDL). The parsing process generates a parse tree for the executable statements and a symbol table. Symbol table constitutes names and types of design parameters, which involve input, output, and internal variables, that reflect design specifications.

2. Layout To Logic tool (LTL) [4]: LTL is applied to design implementation in a form of *SPICE* file, to generate a formulae reflecting desired layout functionality. The formulae are logic expressions represented by propositional logic binary trees.

3. Interface environment for the set of design variables, including:

   - Symbolic Arithmetic Solver (SAS): Arithmetic expressions involved in design specifications are applied to SAS to generate binary equivalent expressions.
   - Propositional Solver (PS): To manipulate propositional logic expressions involved in the design. One of its main components is the Disjunctive Normal Form (DNF) converter which generates a unique and minimal formula (sum of product) for any given function. DNF n-ary trees of Specifications Binary Trees, DNFSBT, are generated for all formulae involved in design specifications including logic and conditional expressions. The same transformation process is carried out for binary trees of design implementation. That produce m-ary trees in DNF, DNFLBT, from the Layout Binary Trees, (LBT).
   - Compare: DNFLBT and DNFSBT are compared for equivalence, implication or contradiction, using compare tools. Verification results indicate whether the given DNF trees are isomorphic (totally matched) constitutes implications, or contradiction.

The introduced system verifies design layout against design specification. The methodology is to bring the layout description up into an abstract form of propositional logic. The specification is decomposed from HDL into the logic level of abstraction to meet with that of the implementation. The two abstractions are then compared for implication if not equivalence. Both abstractions are represented as binary trees of propositional expressions. These expressions are simplified into Disjunctive Normal Form to accelerate equivalence checking.

2 Interface Environment

Binary trees generated from layout are randomly constructed from transistors and connectors into

NAND, NOR, or even XOR boolean expressions. The order and the structure of the generated expressions require further refinement to facilitate and accelerate verification process. Also, the semantics of the generated parse trees may not be totally logic . It might include arithmetic, conditional, looping and other constructs. Therefore, transformation tools are added to convert parse tree expressions into its logical equivalence.

2.1 Symbolic Arithmetic Solver

Expressions involved in the parse trees are transformed into logical expressions through the Symbolic Arithmetic Solver (SAS). It includes set of routines to transform arithmetic expressions into its logical equivalence. That includes, Binary adder, subtracter, multiplier, and negation.
Let $a_i \in A$, and $b_i \in B$, where A, and B are two arrays of size n and m respectively. And let R be an array of a size dependent on n, m and the operation to be carried out. The following are sample algorithms of the SAS:

BINARY ADDER(A,B,n,m): R;
begin
$c \leftarrow nil$
If $n \geq m$ *then*
$\forall\, a_i \in A$ *and* $b_i \in B$, *i=1..m do*
$ri \leftarrow XOR(a_i, b_i, c)$
$c \leftarrow (c\ OR\ a_i\ AND\ b_i)\ AND\ (a_i\ OR\ b_i)$;
$\forall a_i \in A$, *i=m+1..n do*
$r_i \leftarrow XOR(c, a_i)$
$c \leftarrow a_i\ AND\ c$;
$r_{n+1} \leftarrow c$
else $R \leftarrow$ *BINARY ADDER(B,A,m,n);*
end.

BINARY Multiplier(A,B,n,m):R;
begin
Row $\leftarrow$ *Generate Multiplicand rows(A,B,n,m)*
$D \leftarrow Row_0$
$R_0 \leftarrow D_0$
$\forall j,\ (1 \leq j < min(n,m))$ *do*
Shift-Right(D)
$D \leftarrow BINARY\ ADDER(Row_j, D,$
max(n,m),max(n,m)-j)
$R_j \leftarrow D_j$
end.
Generate Multiplicand rows(A,B,n,m):Rows;
begin
if $n \geq m$ *then*
$\forall\, b_i \in B$, *i=1..m Do*
$\forall\, a_j \in A$, *j=1..n Do*
$Row_{i,j} \leftarrow b_i\ AND\ a_j$
else
Rows $\leftarrow$*Generate Multiplicand-rows(B,A,m,n)*
end.

Where *AND*, *OR* and *XOR* are the operators for logical product, logical sum and logical exclusive sum, respectively, D is a one dimensional array, *Row* is a two dimensional array, *"max"* and *"min"* are functions that return the maximum and the minimum of two given values, respectively, and *"Shift-Right"*, is a simple function that shifts a given bit array one bit right. Two's complement subtraction is implemented in the subtracter algorithm, it has the same structure and almost same steps as the Binary adder except that the subtrahend variable is negated. Negation algorithm *"Negate"*, constitute pre-appending the negation operator *"NOT"* to a given expression (subtree). It is applicable for simple and array variables.

2.2 Propositional Solver

All logic expressions generated by the SAS and LTL tools are manipulated to generate uniform expressions in DNF.
Propositional Solver, (PS), tools includes set of routines to manipulate propositional binary tree expressions. It provides tools to combine subtrees, with different operations (ANDing, ORing, XORing and Negating); to decompose a tree into subtrees and to transform a tree into different patterns (e.g., pure NAND, or pure NOR expressions). DNF converter, is the most

useful tool for our application. As shown in Fig., 1 a propositional expression is presented as a binary tree, the variables are the tree leafs, while the root and internal nodes are boolean operations. Each subtree root (boolean operator) has left and right subtrees corresponding to its left and right operands. Negation operator *"NOT"* has a single subtree as its next operand.

DNF Algorithm:

The first step in the DNF algorithm is to generate a tree that has the negation operator as a root for only leaf nodes. This operation is performed by the procedure *"Negation Bubbling"*, which bubble all negations up to the leaf nodes. It *traverse* a given binary tree to generate a bubbled binary tree. Let $t_i \subset T$, where T are the set of all binary tree expressions. And let Nj_{t_i} be the set of nonterminal nodes (root or internal node) $\in t_i$ where j=1..3, corresponding to the operators *"NOT"*, *"AND"* and *"OR"*, respectively, n_{t_i} be the set of all terminal nodes (variable names) $\in t_i$, and $tn_i \in \{n_{t_i}, Nj_{t_i}\}$. The following is the algorithm for *Negation Bubbling*:

Traverse(T):T
begin
$\forall\, t_i \in T$ *do*
if $tn_i \in n_{t_i}$
then $t_i \leftarrow n_{t_i}$
else if $tn_i \in Nj_{t_i}$
then if $i = 1$ *then*
$t_i \leftarrow Bubble(tn_i)$
$t_i.left \leftarrow traverse\ (tn_i.left)$
$t_i.right \leftarrow traverse(tn_i.right);$
end.

Bubble(t_i) : t_i
begin
if $tn_i.next \in n_{t_i}$ *then* $t_i \leftarrow tn_i$
if $tn_i.next \in Nj_{t_i}$ *then*
case j of
1 : $t_i \leftarrow tn_{t_i}.next.next$
2 :t_i $\leftarrow$*creat-or(Negaye*($tn_{t_i}.next.left$),*Negate*($tn_{t_i}.next.right$))
3 : $t_i \leftarrow$*creat-and(Negate*($tn_{t_i}.next.left$),*Negate*($tn_{t_i}.next.right$)
Traverse(tn_{t_i});
end.
The idea of *Negation Bubbling* algorithm is to recursively convert each negated expression (subtree) into its equivalent normal form, by applying DeMorgan's laws (e.g., *NOT(A AND B) = NOT (A) OR NOT (B))*.

The second step in DNF procedure is to generate the sum of product of the given bubbled tree as follows:

Traverse SOP(t_i) : t_i
begin
if $tn_i \in Nj_{t_i}$ *then*
Traverse SOP($tn_i.left$)
Traverse SOP($tn_i.right$)
if $tn_i \in Nj_{t_i}$ *then*
case j of
1 : $t_i \leftarrow Negate(pop(n_i))$
2 : $t_i \leftarrow Product(po(n_i), pop(n_j))$
3 : $t_i \leftarrow Concat(po(n_i), pop(n_j))$
push(t_i);
end.

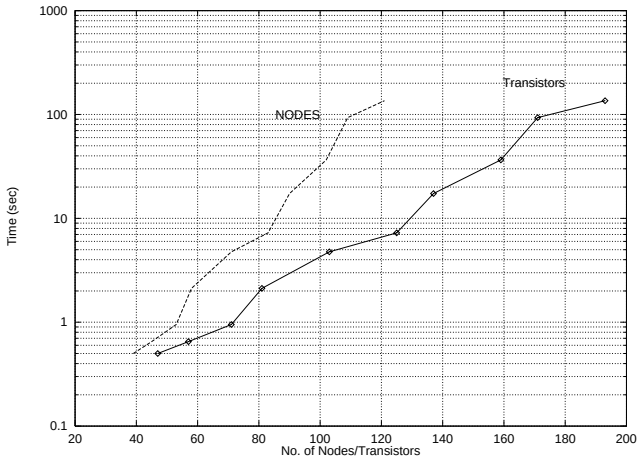

Figure 2: Number of transistors & nodes versus the execution time

A stack structure has been used with the two operations *push* and *pop*. Processed subtrees

are pushed into the stack one at a time, then poped pack for further operations. *"Product"*, is a procedure to compute the logic product of two subtrees. While, *"Concat"*, is a routine to combine subtrees with the same operation (*AND* and *OR* only). The resulted DNF tree is k-ary tree with disjunctive, (*OR*) operator root, where k is the number of created conjunctive expressions. The root is linked to k subtrees each with a conjunctive, (*AND*) operator root. Each of these roots has a subtree of leaf nodes (simple terminal). DNF tree leafs represent terminal nodes in the circuit design, each of which has a designated pointer. Node and subtree pointers are generated with no repetition. This unique representation has been carried on through all the procedures to minimize storage size and facilitate variable referencing.

3 Conclusion

The presented formal verification system is successfully applied to a set of arithmetic circuits (full adders, subtracter, comparator, and multipliers, with different input array sizes). The results for both adder and subtracter circuits were isomorphic (exact matching of abstractions).
On the other hand, satisfiability was the result for both multiplier and comparator (i.e., the implementation was contained or implied in the specification). That is, because The symbolic solver does not necessarily generate a matching expression for the specification, and, also, the layout synthesiser performs simplification and optimization on circuit layout. It is a safe approach not to simplify symbolic specification because it might result in a false rejection of a correct design. System Performance is shown in Fig. 2. We claim that such performance can be improved, because, constructing the logic from the layout is totally independent of parsing design specification, therefore the two processes can be executed in parallel to generate the design logic.

References:

[1] M. Fujita, Y. Matsunaga, and T. Kakuda. *"Automatic and Semiautomatic verification of switch-level circuits with temporal logic and binary decision diagrams,"* In ICCAD, pp. 38-41, 1990.

[2] K. L. McMillan, *"Symbolic Model Checking,"* Kluwer Academic Publishers, Boston, 1993.

[3] Fredrick J. Hill and Gerard R. Peterson, *"Introduction to Switching Theory and Logical Design,"* John Wiley & sons, New York, 1974.

[4] Fatma A. El-Licy, and Prof. Hoda S. Abdel-Aty-Zohdy, *"Verification System Interface for VLSI Combinational Circuits,"* 41^{st} IEEE Midwest Symposium on Circuits and Systems, pp. 408-411, Aug 1998.