VLSI Design
1994, Vol. 1, No. 3, pp. 243–259
Reprints available directly from the publisher
Photocopying permitted by license only

Block-Level Logic Extraction from CMOS VLSI Layouts

INDERPREET BHASIN and JOSEPH G. TRONT
Department of Electrical Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA

(Received August 30, 1990, Revised February 21, 1991)

This paper describes a Prolog based Block Extraction System (ProBES) which converts a transistor level description of a CMOS circuit into a logic block level description. The operation of ProBES is conceptually similar to that of a circuit extractor. However, whereas a circuit extractor is used to identify circuit primitives such as transistors, resistors and capacitors from the geometrical information in a mask level layout description, ProBES can be used to identify predefined gates and logic blocks in a CMOS transistor network. ProBES operates according to the circuit hierarchy. Basic gates such as inverters, transmission-gates, nands, nors, etc. are identified first. Logic blocks composed of these gates are then identified. More complex blocks which contain blocks already identified are recognized next and so on. ProBES is meant to be used as an aid in the verification of logic design. It can provide a connectivity check for a circuit.

Key Words: *Layout Verification; Logic Extraction; VLSI Design*

INTRODUCTION

A significant aspect of the VLSI design process is verifying that the final layout correctly represents the intended logic. Errors can arise due to the manual effort involved in translating logic designs to physical structures in silicon. Digital circuits have an implicit hierarchy in their description. However, it is easier to verify a circuit at a higher level of description when it is pruned of unnecessary details. A desirable approach in verifying a design would therefore be to transform a detailed description into a less detailed one [1]. A system has been implemented to automatically extract logic blocks from a CMOS transistor level description. This process of block extraction is one in a series of progressive steps toward translating the circuit description to a higher level.

The description of the circuit is assumed to be available at the transistor level. Several schemes already exist to extract transistor level interconnectivity information from the mask level layout [2, 3]. Starting with the description generated by a circuit extractor, we derive a higher level description in the form of an interconnection of functional blocks. Previous work of a related nature has concentrated on the extraction of very basic gates, based purely on topological comparison with reference gates [4, 5, 6]. A generalized approach is outlined in [7, 8] for checking the logical correctness of FET designs through circuit recognition. In [9] a block extraction scheme is described to extract predefined circuit blocks from a SPICE like network description. But, the recognition of blocks is based purely on circuit topology and is accomplished through comparison of reference graphs of logic functions with derived graphs in a circuit. A shortcoming of methods which rely on a graphical approach is that a test for graph isomorphism establishes topological, not functional equivalence. As a result, two implementations of the same logic function may be considered different. This imposes a restriction on the designer to create designs which retain a topological correspondence with the original designs. A logic extractor described in [10] translates CMOS transistor level descriptions to gate level in order to enable gate level logic simulation. The procedure involves combining transistors into series and parallel branches. By combining these branches, the logic function is established. An advantage of this method is that no user reference logic functions are needed. However, by this process only primitive logic functions or logic blocks composed of

these primitive functions can be extracted. This is a limitation because designers often use non-standard logic gates and not all logic blocks can be described as combinations of simple primitive gates. In ProBES, any arbitrarily defined logic block can be extracted. An array of predefined gates is recognized from a transistor level description of the circuit, based on the transistor level structure of a gate or the symbolic boolean expressions at the output node of a gate. The extraction of these boolean expressions is part of the program. Following the recognition of basic gates, functional blocks composed of these gates can be recognized. In the program, a library of functional blocks is maintained. Within this library, a logic block is defined as a set of Prolog clauses. More than one description of the same type of logic block is possible. The recognition of a functional block becomes equivalent to satisfying a goal, subject to the constraints specified in the rules for that block. These constraints take a structural form wherein a block is defined as a composition of lower level blocks and gates. The basis of block extraction is the circuit topology as well as the deduced logical behavior of circuit blocks. Unlike previous methods which take a purely graphical approach, we use the symbolic features of Prolog to define functional blocks. The instances of these blocks in a circuit are then recognized using the pattern matching ability inherent in Prolog.

THE BLOCK EXTRACTION APPROACH

Currently, simulation of the designed circuit is the predominant method of verification [11]. For large circuits, complete verification is not possible even with switch level simulators. Other than simulation, verification methods based on topological comparison [4, 5, 6, 9] have been suggested. But, these methods have limitations and are not generally applicable. In the absence of simulation, the functional correctness of a circuit has to be deduced from a purely static analysis based on the physical connectivity of the circuit components. The task of establishing the functional correctness of a circuit is easier when the description of the circuit is available at a higher level in the abstraction hierarchy.

We describe a method to recognize logic functional blocks from a transistor level description of a circuit. Translating a circuit description consisting of an interconnection of transistors to a description consisting of an interconnection of logic blocks is a step toward raising the level of abstraction. We may observe how such a recognition process can enable verification at a higher level. Assume that a circuit is specified as a composition of subcircuits. A recognition procedure operating on the implemented design identifies each subcircuit that makes up the larger circuit. Now, we can check whether these subcircuits are configured in a way that realizes the intended circuit.

An intelligent program attempting to recognize logic blocks in a circuit would need a knowledge-base which enables it to do the following:

- Correlate derived boolean expressions with defined functions.
- Match particular circuit topologies to defined functions.

The recognition process then, is based on both, the topology of the circuit and the logical behaviour of the circuit. The broad outline of the recognition and verification procedure can be stated as follows:

1. Circuit recognition rules are maintained in a library. These rules define logic gates in terms of transistor interconnections or in a behavioural form in terms of boolean logic expressions.
2. The transistor level description of the circuit is processed to a form where the application of recognition rules can proceed. This processing step involves partitioning a circuit into smaller subcircuits and extracting logic expressions at output nodes of these subcircuits.
3. Gates in the circuit are extracted based on these logic expressions.
4. The circuit to be verified is specified as an interconnection of its constituent logic blocks. Rules describing each of these logic blocks should be present in the rule library. A logic block is defined in terms of the gates or lower order logic blocks contained in it.
5. Using the recognition rules, an attempt is made to find an instance of each logic block in the original circuit, within the designed circuit. The recognition procedure is constrained by a need to preserve the connectivity between logic blocks as specified in the original circuit. This connectivity pattern is implicit in the input and output node names of the logic blocks.
6. If an instance of each constituent logic block is found, one can conclude that the circuit reflects the desired logic.

In order to implement the described approach, a programming language is desired which facilitates an

easy description of logic circuits as well as the manipulation of boolean expressions. Prolog is a logic programming language well suited to the structural description of digital circuits.

The use of Prolog has been suggested of late for the description and verification of digital circuits [13, 14]. In [15], a Prolog based approach is described for verifying gate level circuits through symbolic simulation. A Prolog based connectivity checker is described in [16]. Prolog allows a circuit to be described in a manner which is easy to understand. This is particularly advantageous when we consider the framing of rules for the recognition of logic blocks. If a new logic block is used in a design, the user can add the rule describing the block, to the rule library with relative ease. There are two major features of Prolog which are very useful in verification. These are the pattern matching ability and automatic backtracking.

Complete details for this block extraction algorithm along with information on the software that implements this approach can be found in [21].

SYSTEM OVERVIEW

The overall scheme for the extraction of gates, blocks, and logic expressions from a layout description (CIF file) is shown in Figure 1. In this scheme, the Magic system [17] is shown, used in conjunction

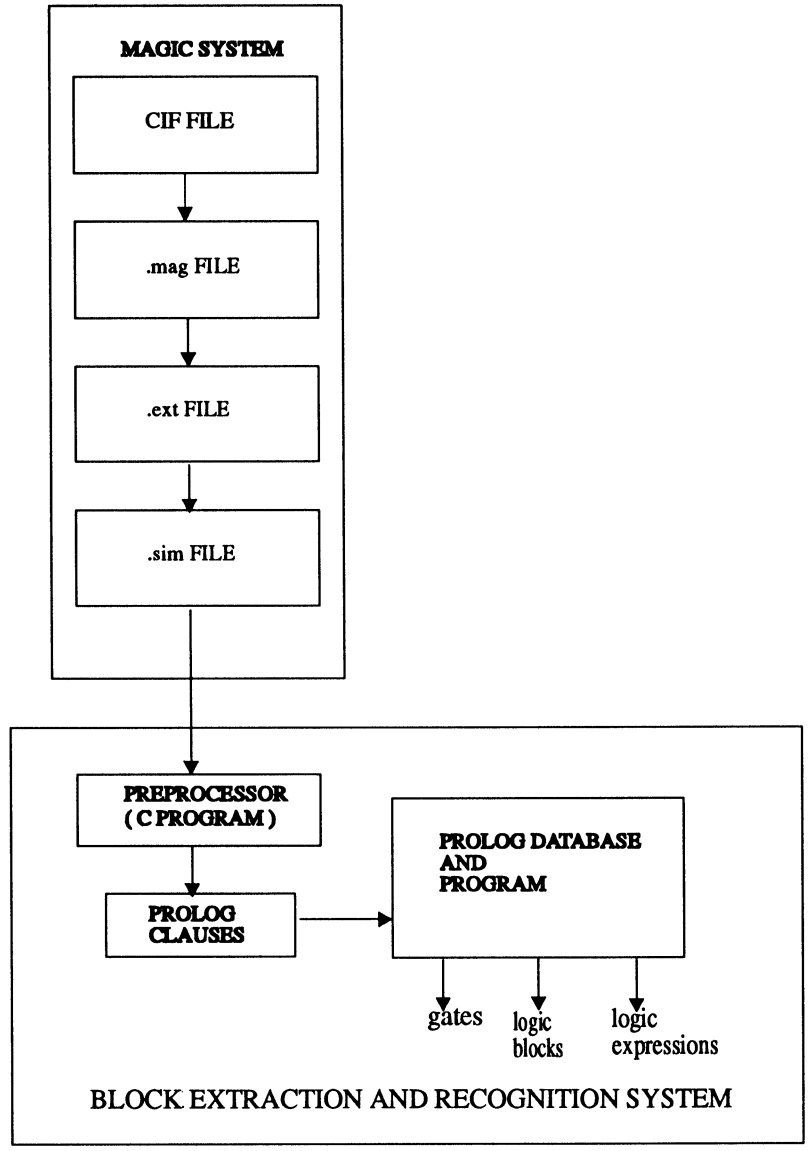

FIGURE 1 A Block Extraction and Recognition System.

with the block extraction and recognition system. Any other circuit extraction procedure which produces a transistor level description of a layout in the Magic .sim format, can be used instead.

A C preprocessing program called sim2pro, which forms a part of the block extraction system, converts a file in the Magic .sim format into a list of Prolog clauses which form the input to the recognition program written in Prolog.

The preprocessing program converts a transistor in the .sim format to the following Prolog clause:

trans(*type*,*g*,*s*,*d*).

where

Type indicates the transistor type, namely n or p

Gate stands for the gate node of the transistor

and *s and d* represent the end terminals of the transistor, namely source and drain. In addition, the program searches through the attribute lists of terminals to identify nodes which are primary inputs and outputs of the circuit. To enable the program to extract this information, the designer when creating a layout needs to affix the labels *input* and *output* to the CIF geometric structure associated with the primary input and output nodes of a circuit. The Magic layout system allows attributes to be associated with individual geometries (usually rectangles for Manhattan style layouts). Therefore, a rectangle (poly, diffusion or metal) which corresponds to a primary input can be labelled as such and this information is retained by the circuit extractor. A generalized form of the input file to the block extraction program is as follows:

input_list([I_1,I_2,...I_n]).
output_list([O_1,O_2,...O_m]).
trans(Type,Gate,T1,T2).
.....
..... .

The input description consists of a list of primary input nodes to the circuit [I_1,I_2,...I_n] a list of primary output nodes of the circuit, O_1,O_2,...O_m followed by a list of transistors in the circuit.

Figure 2(a) shows a CMOS XNOR circuit. The .sim file describing this circuit is shown in Figure 2(b) and the list of Prolog clauses derived by sim2pro from the .sim file is shown in Figure 2(c).

The list of Prolog clauses describing the transistors in the circuit form an input to the Prolog program. The database consists of a list of Prolog functions as well as rules for recognizing gates and logic functional blocks in the circuit. The program also derives logic expressions at output nodes of logic blocks.

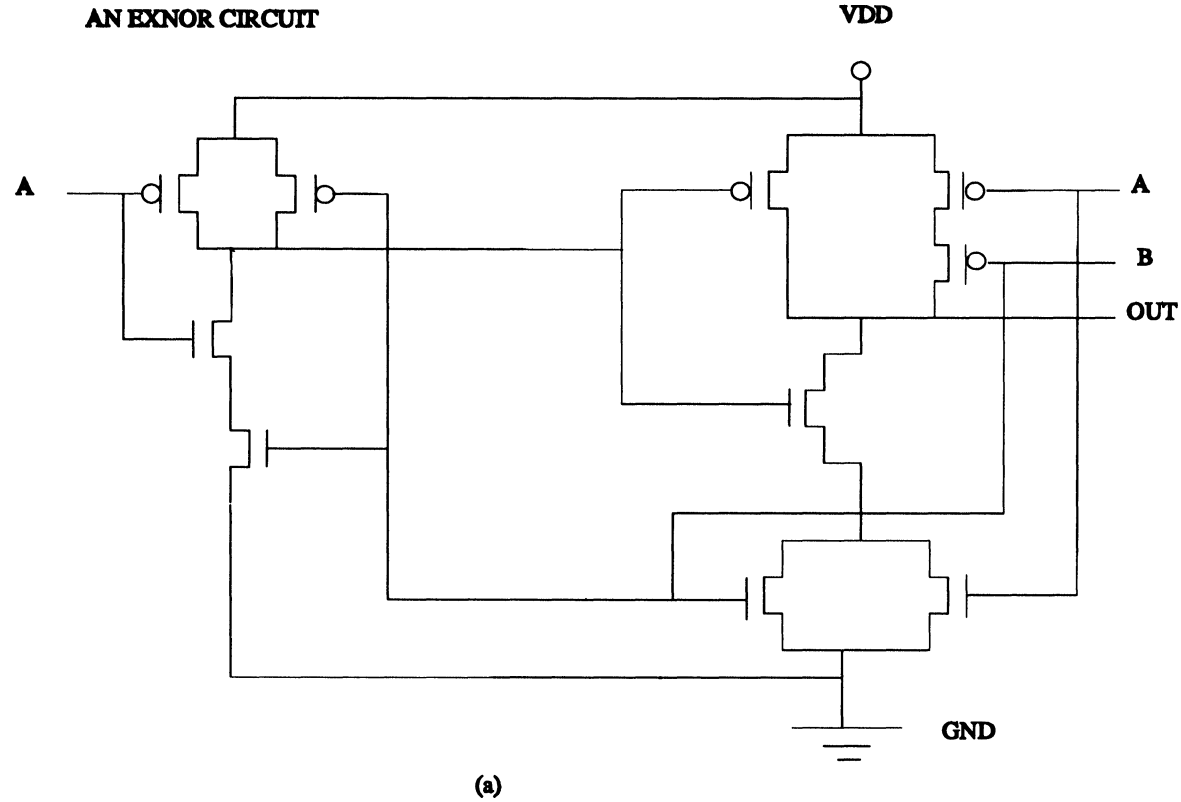

FIGURE 2 A CMOS EXNOR circuit.

EXTRACTION OF CIRCUIT BLOCKS

The extraction of circuit blocks from a transistor level description consists of several steps. These steps are described in this section.

Recognition of Inverters and Transmission Gates

Inverters and transmission gates can be recognized directly from their transistor structures. The early recognition of inverters is necessary to identify complementary signals in the circuit. This information is useful in later steps when logic expressions are derived at the output nodes of partitioned blocks.

Inverters are identified by looking for a pair of p and n type transistors having the same gate input and forming a direct path from vdd to ground. CMOS transmission gates are recognized by identifying a pair of nodes connected to both p and n type edges and having complementary gate inputs. The recognized inverters and transmission gates are inserted into the database. The transistors forming these are then deleted from the database.

Deriving a Node-Oriented Data Structure

The input to the program containing a list of transistors in the circuit describes the circuit in the form of an implicit graph. In order to recognize logic blocks, we need to partition the network described by the transistor netlist. The partitioning step converts the network into smaller subnetworks, the behavior of which can be derived independently.

Each transistor in the circuit stands for an edge in the circuit graph. The input description of the circuit is therefore, a list of edges. Prior to the partitioning procedure, a node-oriented data structure for the graph, consisting of a list of nodes in the circuit along with the connectivity information for each node is derived. This facilitates partitioning.

A node is represented in the following format after processing the transistor list:

node(*Nodename, Nodetype, Edgelist, Gatelist*).

where

Nodename is the name of the node in the circuit.

Nodetype is a variable instantiated in a later step to one of the node categories.

Edgelist is the list of edges incident with the node.

Gatelist is the list of edges to which the given node is a gate input.

Node Classification

Preceding the partitioning of the circuit into smaller subcircuits, the nodes in the circuit are classified into one of the four categories: input, pullup, external, and normal. The terminology has been borrowed from Nmos partitioning techniques. In the case of CMOS circuits, we proceed to define the classification of nodes as follows:

1. *Pullup Nodes:* A pullup node is defined as a node which is a drain or source node of both a p and an n type transistor, in the circuit which results after all transistors forming CMOS transmission gates have been removed. See Figure 3(a). The need to identify all of the transmission gates in the circuit in the first step is evident here. This definition of pullup nodes is merely intended to enable partitioning. It does not always conform to the usual notion of a pullup node in a CMOS circuit as a node connected to vdd via a p block.
2. *Input Nodes:* An input node is defined as one of the following:
   (a) A primary input node in the circuit.
   (b) A primary output of the circuit which is not also a pullup node.
   (c) A node which is a drain or source node of a transistor, but is not a pullup node and is connected to an end terminal of a transmission gate. (Figure 3 (b)).
3. *External Node:* An external node is defined as one of the following:
   (a) A node which is connected only to gates of transistors and is an end terminal of a transmission gate. (Figure 3 (c)).
   (b) A node which is not a pullup or input node and is connected to the gate of a transistor (Figure 3 (d)).
4. *Normal Node:* A normal node is one which does not fall into any of the categories above. (Figure 3 (e)).

CIRCUIT PARTITIONING

The circuit partitioning technique formulated here is similar to the partitioning methods used in the presimulation phase in switch-level simulators. One such partitioning method for Nmos circuits is described in [18]. In a CMOS circuit the broad idea behind the partitioning procedure is to divide the set of transistors in the circuit into driver, load and pass transis-

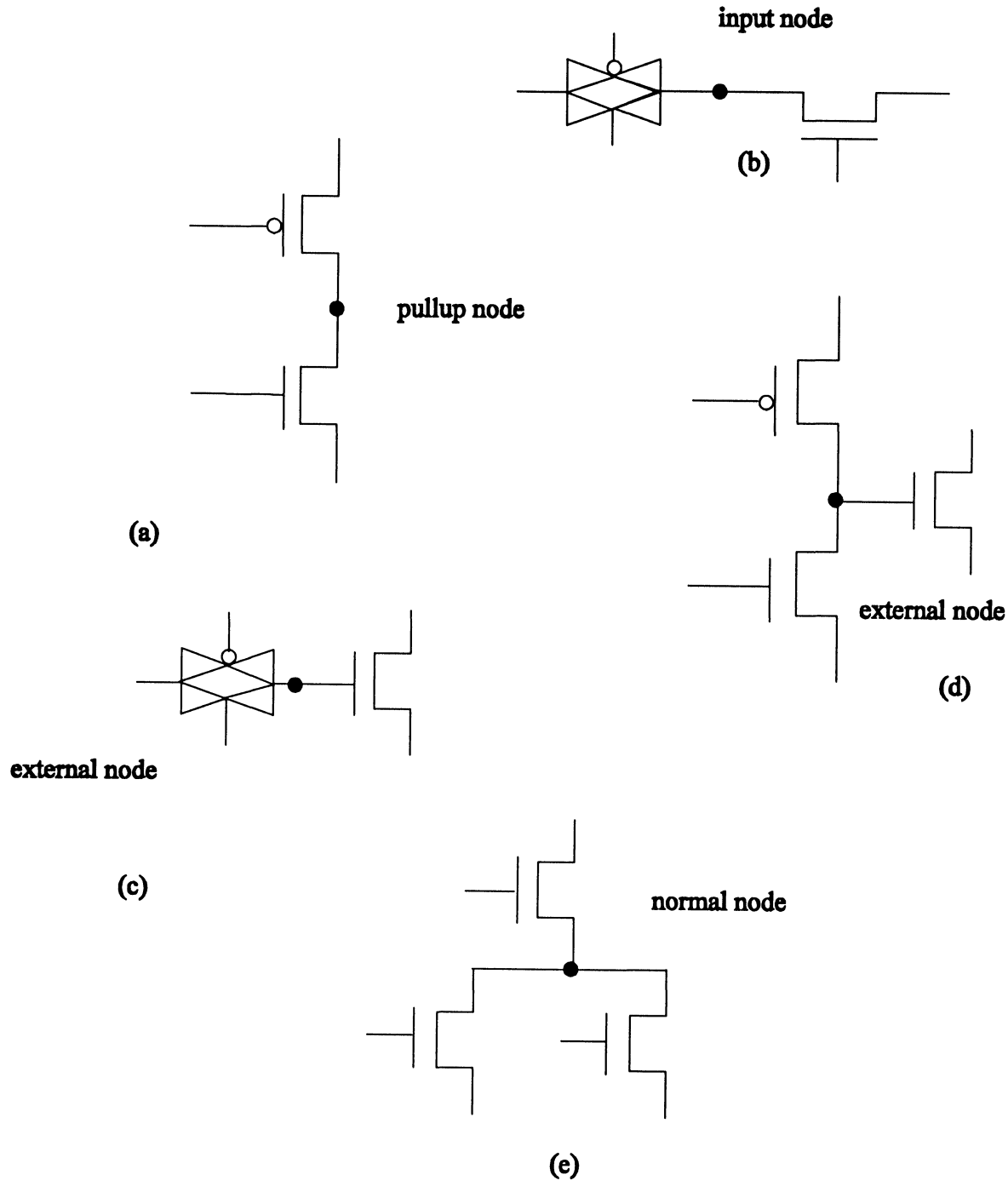

FIGURE 3 Classification of Nodes.

tors. Driver transistors are a group of n type transistors which could form the n block of a CMOS logic block. Load transistors are a group of p type transistors which could form the p block of a CMOS logic block. The remaining transistors are grouped into pass transistor blocks.

Partitioning is accomplished by *splitting* the circuit graph at boundary nodes. These boundary nodes consist of vdd, ground, input nodes, and pullup nodes. Splitting the circuit graph at a vertex means dividing the vertex into two or more vertices, so that all the edges incident with the vertex become mu-

tually disconnected at that vertex. We seek groups of transistors bounded by boundary nodes. The splitting procedure is implemented as a search starting at a boundary node and terminating at all boundary nodes reachable from the starting node.

DERIVATION OF LOGICAL EXPRESSIONS

Once the partitioning is complete, transistors in the circuit have been grouped into sets of load, driver, and pass blocks. Load and driver blocks are multi-input-single-output logic blocks, whereas pass blocks may be multi-input-multi-output logic blocks. The next step consists of deriving the logical expressions at output nodes of logic blocks in terms of the inputs to the blocks. Approaches towards the automatic generation of Boolean expressions in a MOS network are described in [19, 20].

In out technique, a logic expression at an output node of a logic block is formed by tracing paths from the node to all *reachable driving signal* nodes of the block. A node n_j is reachable from a node n_i if there is a path in the circuit graph that connects the two. Nodes that fall into the category of driving signal nodes are vdd, ground and input nodes to the block.

At each output node of a block, two logic expressions are generated to represent the logical high and low value at that node. For an output node V of a logic block having inputs $I_1,I_2,\ldots I_3$, the following clause is inserted into the database:

value(V,[$I_1,I_2,\ldots I_3$],V_Low,V_high)

where V_low and V_high are the boolean expressions for the logical 0 and 1 values at the node V, respectively.

The logic expressions are derived in the sum of products form. Each path traced from the output node to a reachable driving signal node yields a product term and the OR of all such terms gives the sum of products form for the expression.

LOGIC CLASSES RECOGNIZED

A load-driver logic block is formed by combining a load block and a driver block which have the same output node. In order to derive the logic expressions at the output node V of a load-driver block, we need to consider all the paths leading from V to vdd and gnd. We represent the generated logic expressions for a load block output as:

$value(V,1,I_l,T_l)$

and for a driver block output as:

$value(V,0,I_d,T_d)$

where T_l and T_d are the logic expressions derived by considering paths to vdd and gnd respectively, and $I_l(I_d)$ are the inputs to the load (driver) block. A load-driver block is recognized by identifying a pair of clauses as above, which have the same first argument V, (the same output node) and complementary second arguments (1 and 0).

Three classes of logic, namely *pseudo - NMOS, clocked CMOS,* and *fully complementary CMOS* are recognized. The rules for recognizing the logic category are listed below:

1. If T_l = [[1]], the load block consists of a single p type transistor with a grounded gate, and the logic is pseudo - NMOS. In this case, V_low = T_d and V_high = $\overline{(T_d)}$ The logic expression for the output node V is inserted into the database in the form:

   value(V,I,V_low, V_high)

   where $I = I_d$.

2. If there is a boolean variable c present in each product term in T_d and its complement $\bar{c}$ is present in each product term in T_l, then we have clocked logic. In this case c is removed from each product term in T_d to get T_{dll} and $\bar{c}$ from each product term in T_l to get T_{l1}. The logic expression at the output node V is asserted as:

   value(V,clocked,c,I,V_low,V_high)

   where $I = I_d - \{c\}$, $V\_low = T_{d1}$ and $V\_high = T_{l1}$.

3. If $I_l = I_d$, we consider the logic to be fully complementary. In this case $V\_low = T_d$ and $V\_high = T_l$. The logic expression at V is represented as:

   value(V,I,V_low,V_high)

   where $I = I_d$.

In case 1 above, the complement of a boolean expression has to be derived. The program has the ability to find the complements of boolean expressions. The procedure however, is exponential in time complexity with the number of variables in the expression to be complemented. This means that load-driver blocks with a large number of inputs would restrict the program speed.

PASS AND TRANSMISSION BLOCKS

Transmission gates and pass blocks in the circuit have nodes which have been labelled as both inputs as well as outputs of these blocks. Such nodes are denoted as io nodes. and have to eventually be resolved as either input or output nodes of these blocks and gates. We do this by tracing the fan-in and fan-out of nodes which form the interface nodes between blocks.

In the case where the directionality of a transmission gate is unresolved, it is left as bidirectional.

RECOGNITION OF GATES

The term *gate* as used here implies a collection of transistors with one or more inputs and only one output. The program database contains a list of user defined gates. Each defined gate has an associated set of attributes which include: the name by which the gate is referred to in the program, the inputs and output of the gate, and two boolean equations which express the logical high and logical low values at the gate output in terms of the inputs. An additional attribute may be present describing the gate as being clocked, in which case the clock signal would also be specified.

The reference boolean equations associated with a gate are in terms of Prolog variables. (In the version of Prolog used, variables begin with upper-case letters. Variables may be instantiated to constants which begin with lower-case letters.) The derived expressions at the output node of a logic block are in terms of Prolog constants. These constants are, in effect, the node names of the inputs to the logic block. As an illustration, the logic at the output node y of a block implementing the exor function and having inputs a and b is represented as:

value(y,[a,b], [[a_, b_],[a,b]],[[a_,b],[a,b_]]).

In the example above, the boolean variables: a, a_,b, and b_ are all constants. In the program database, a two input exor gate may be defined using the following attributes:

gatename: exor2
Inputs: [A,B]
High expression: [[A_,B],[A,B_]]
Low expression: [[A_,B_],[A,B]]

where A and B are Prolog variables representing the two inputs to the exor gate. The variables A_ and B_ would be related within the rule to A and B by the clauses below:

complement(A,A_).

and

complement(B,B_).

The identification of a gate in the circuit becomes a problem of finding a correspondence between an extracted logic expression set •• the pair (v_low,v_high) for an output node v and a reference gate with an associated set of logic expressions. In order to do so, the definitions of n input reference gates are examined, where n is the number of inputs in the expressions for the output v. An equivalence is sought to be established between the derived logic expression set and the reference logic expression set of an n input gate.

The following equivalence relationship is used:

F(x) and G(x) represent the same boolean function if

$F(x)\overline{G(x)} + \overline{F(x)}G(x) = 0$ Or in other words
$F(x)\overline{G(X)} = 0$ and
$\overline{F(x)}G(x) = 0.$

The above equations imply that if F(x) = 1, then G(x) = 1 and if G(x) = 1 then F(x) = 1. This in turn implies that the truth tables for the two functions match. The stated equivalence is valid however, only in the case when the truth tables for F and G contain no 'don't cares' in the output column. The presence of don't cares leaves the function incompletely specified. If don't cares are present, there would be a likelihood of the function being incorrectly recognized as shown below. This, unfortunately, is a restriction on the method.

The equivalence relationship described above is applied in the recognition procedure. Assume that in the gate library we have a gate 'f' defined by the boolean expressions f_0 and f_1 which represent the logical low and high values of the gate output, respectively. An extracted expression 'g' represents a gate 'f' if:

$$f\_0 * g\_1 = 0 \quad (1)$$

and

$$f\_1 * g\_0 = 0 \quad (2)$$

For a clocked inverter with inputs clk and x the extracted expression 'g' is defined by: $g\_1 = clk\bar{x}$ and $g\_0 = clkx$. In this case because the output is undefined when clk is low, by the equivalence relationship stated above, the function could be incorrectly recognized as an exclusive-or for which the

defining expressions are $f\_1 = clk\bar{x} + \overline{clk}x$ and $f\_0 = clhx + \overline{clk}x$.

As stated before, the boolean expressions describing a gate in the library are in terms of Prolog variables. Before we can attempt an equivalence through Eqs. (1) and (2), these variables have to be instantiated as symbolic constants in the logic expression which we wish to recognize.

As an illustration, the exor gate described earlier uses the prolog variables A and B. The derived expression is in terms of Prolog constants a and b. A one to one assignment from the set {a,b} to the set {A,B} has to be made. In this case, since the inputs to an exor are symmetric, either of the assignments (A=a,B=b) or (A=b,B=a) would prove or disprove the equivalence. In the general case we have derived expression which is in terms of the n constants $(a_1,a_2,\ldots a_n)$. The boolean expressions for n input gates in the library are in terms of the n variables $(A_1,A_2,\ldots.A_n)$. We seek a one to one assignment of the set $\{a_1,\ldots a_n\}$ to the set $\{A_1,\ldots.A_n\}$. There are n! such assignments. This means that in the worst case, there will be n! steps to prove equivalence before an equivalence test fails. As the number of gate inputs grows, the process of recognition becomes more time consuming. However, it is to be noted that the definition of a 'gate' as applied here refers to a partitioned block within a larger circuit. We can expect that the number of inputs n to a partitioned block would, barring exceptional cases, be a small value. Further, for a large class of gates, we can put the symmetricity or interchangeability of inputs to advantage here. If p of the n gate inputs are interchangeable, then effectively, the number of assignments in the discussion above reduces to $n!/p!$.

The three examples to follow illustrate the gate recognition procedure.

The clause expcomp(Expression_1,Expression_2) in the examples, is true if the following equality holds:

Expression_1*Expression_2 = 0

where both Expression_1 and Expression_2 are boolean expressions.

In the examples above, Expression_2 is the derived boolean expression whereas Expression_1 is the reference expression. The logical high and low expressions of the reference gate appear implicitly in the two expcomp( ) clauses. Functional equivalence is established if the two expcomp( ) clauses are true.

Notice that in the first two examples, the inputs to a nor gate and a nand gate are interchangeable. However, in the case of Example 3 which refers to the gate shown in Figure 4 the inputs are not interchangeable. The clause:

perm(List_1,List_2)

permutes the elements of List_1 to produce List_2. By backtracking all the possible permutations will be tried in order to establish a functional equivalence. The last clause, assert(gate_name(I1,I2...In,O)) asserts the gate by the name gate_name into the data base. I1, I2... In represent the input signals to the gate while O is the output signal.

Example 1.

```
recog(Y,[A,B],Y0,Y1,nor2):-
   expcomp([[A],[B]],Y1),
   complement(A,A_),
   complement(B,B_),
   expcomp([[A_,B_]],Y0),
   assert(nor2(A,B,Y)).
```

Example 2.

```
recog(clocked,Clock,Y,[A,B],Y0,Y1,nand2_
clocked):-
   expcomp([[A,B]],Y1),
   complement(A,A_),
   complement(B,B_),
   expcomp([[A_],[B_]],Y0),
   assert(nand2_clocked(A,B,Y,Clock)).
```

Example 3.

```
recog(Y,[I1,I2,I3,I4],Y0,Y1,and332or3):-
   perm([I1,I2,I3,I4],[A,B,C,D]),
   expcomp([[A,B,C],[A,D,C],[B,D]],Y0),
   complement(A,A_),
   complement(B,B_),
   complement(C,C_),
   complement(D,D_),
   expcomp([[A_,B_],[A_,D_],[C_,B_],
   [C_,D_],[D_,B_]],Y1),
   assert(and332or3(A,B,C,D,Y)).
```

The goal statment in the recognition clause has the format:

recog(Y,Input_list,Y0,Y1,gate_name)

where Y is the output node of a block we wish to recognize, Input_list is the list of input signals to that block, Y0 and Y1 are the logical low and high expressions derived at the node Y in the logic expression generation step, and gate_name is the name by which the gate is referred to in the system.

In the case of clocked logic, two additional parameters, "clocked" indicating clocked logic and "Clock" being the clock signal are present.

In order to handle the interchangeability of gate inputs we maintain rules in the database by which

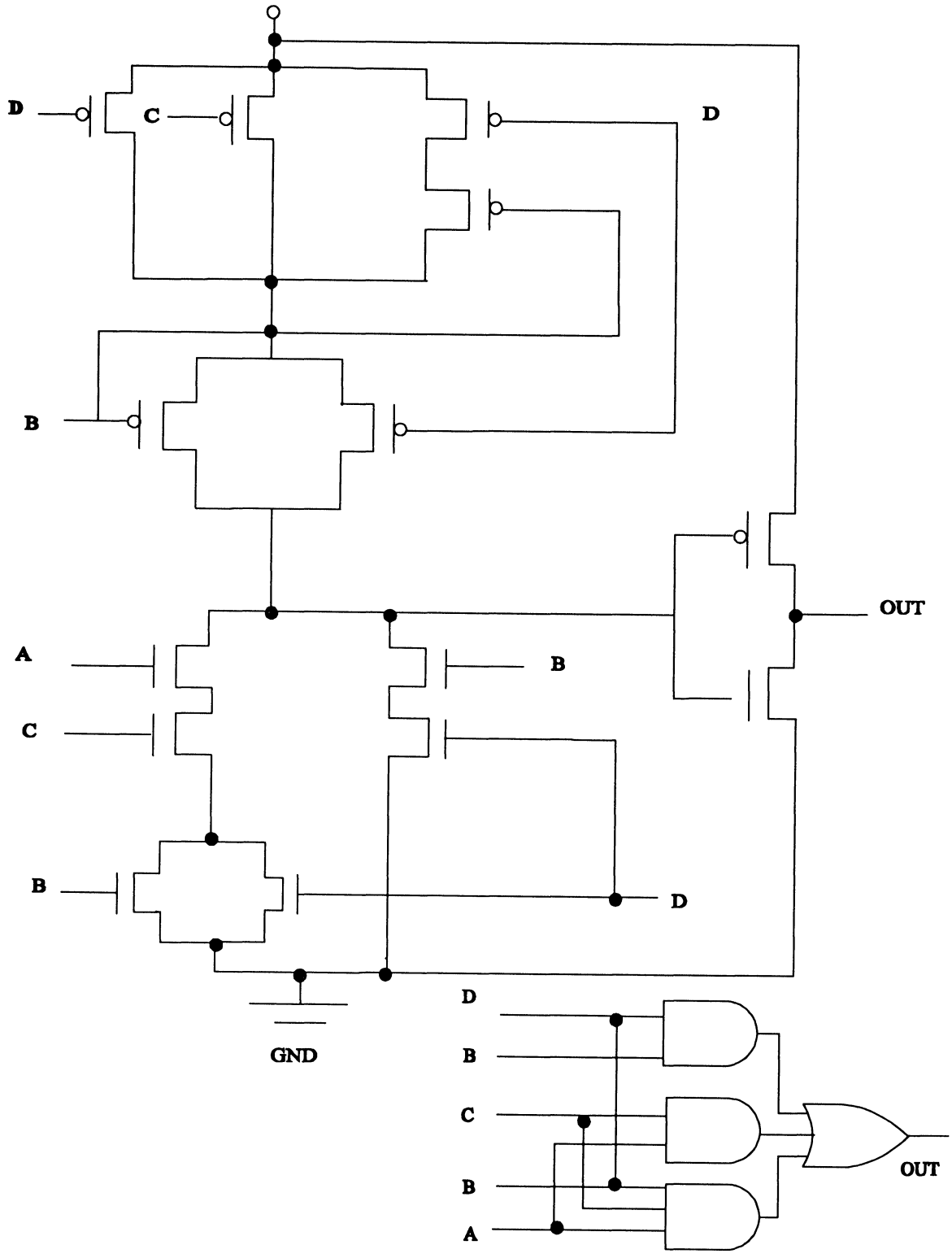

FIGURE 4 An Example Circuit: 'and332or3'.

each gate is related to its various functionally isomorphic forms. The following examples illustrate the idea:

Example 1

```
norgate_2(A,B,Y):-
  nor2(A,B,Y);
  nor2(B,A,Y).
```

Example 2.

```
nandgate_3(A,B,C,Y):-
  perm([A,B,C],[A1,B1,C1]),
  nand3(A1,B1,C1,Y).
```

The rules described above are representative of a general method which can be used to recognize all the logic blocks in a circuit for which corresponding gates exist in the gate library. In this approach we recognize a gate by the logic it implements rather than by the topology as is done in previous gate recognition programs which use graphical techniques. This is an advantage because different implementations of the same gate can be recognized without having to store knowledge of all the different forms.

RECOGNITION OF FUNCTIONAL BLOCKS

Once all the gates have been recognized, the program proceeds to recognize all of the functional blocks which may be composed of these gates. At this level, recognition is through the connectivity of the basic gates. We do not attempt to derive the behaviour of higher level logic blocks.

The descriptions of functional blocks used in a design are present in the program database in the form of rules. A functional block is defined in rule form in terms of the gates it contains and their interconnections. More than one description of the same function is possible.

In Prolog terms, a functional block is defined as a set of clauses. The following rule for a functional block describes a full-adder which is shown in Figure 5.

```
function(full_ad-
der,[[inputs,A,B,Cin],[sum,Sum],
[carryout,Cout]]:-
  exorgate(A,B,S1),
  exorgate(S1,Cin,Sum),
  nandgate_2(A,B,S2),
  nandgate_2(A,Cin,S3),
  nandgate_2(B,Cin,S4),
  nandgate_3(S2,S3,S4,Cout).
```

The recognition of a functional block is equivalent to satisfying a goal subject to the constraints specified in the rules for that block. In the case above, this particular full-adder is found in the designed circuit if all the gates that compose it are found and are interconnected in the manner specified in the rule.

The recognition process involves a search in the database, for components which are specified in the description of a functional block. A functional block may be described both in terms of basic gates as well as other functional blocks. For instance, the multiplier cell of Figure 6 uses the full-adder of Figure 5 and D flip-flops of the kind shown in Figure 7 and is described as follows:

```
function(mult_cell,[[inputs, M1,M2,Pin,Rb],
[clocks,Clk1,Clk2],[outputs,M1out,Pout]]):-
  dffr(M1,Rb,Clk1,M1out,Q1b),
  dffr(Pin,Rb,Clk2,S4,Q2b),
  dffr(Cout,Rb,Clk2,S5,Q3b),
  nandgate_2(M1out,M2,S2),
  inv(S2,S3),
  full_adder(S3,S4,S5,Pout,Cout).
```

The gates and functional blocks recognized from the transistor level description of the multiplier circuit are shown in Figure 8.

The search during the recognition of a functional block is successful when the last clause in its rule is satisfied. In other words, a block is recognized when the last component contained in its structural description is identified in the given circuit.

A clause in the rule for a functional block is constrained by the node interconnection assignments made in previous clauses in the rule. If the search procedure in the recognition of a particular block fails, all the node interconnection assignments done prior to the step at which failure occurs must be undone and alternative assignments explored. The advantage of automatic backtracking is realized here. The procedure seeks all the possible alternatives in terms of the component interconnections to satisfy the recognition rule. If the rule fails completely, an instance of that particular functional block does not exist in the circuit and all node assignments are undone.

THE SPECIFICATION OF RULES FOR GATES AND FUNCTIONAL BLOCKS

In the program database, a library of gates and functional blocks is maintained. Rules describing new gates and functional blocks can be added to the li-

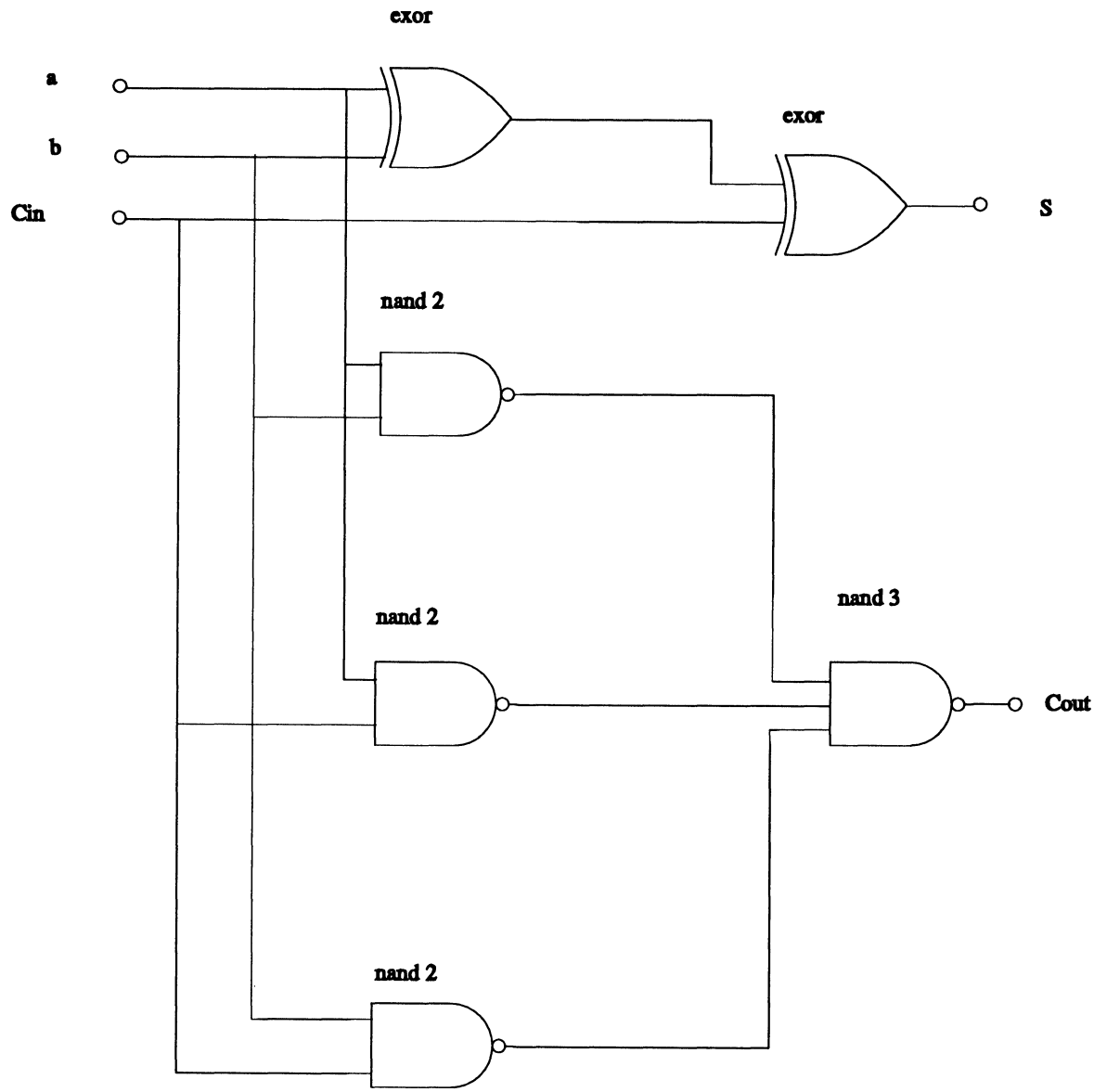

FIGURE 5 A Full-Adder Circuit.

brary by the user. It is to be noted here that the user only needs to follow a specified methodology in adding new rules to the database and is not required to be knowledgeable about Prolog or any Prolog functions used in the system.

The description of gates is in a behavioural form. The gates are grouped in the library by the number of inputs they have. The description of functional blocks is in a structural form. When automatically recognizing gates and functional blocks in a circuit, the program first attempts to find a corresponding gate in the database for each derived logic expression set. After the recognition of gates, the program checks for the possible invocations within the circuit of each functional block defined in the database. Each recognized instance of a functional block is then asserted as a clause in the database.

The recognition of functional blocks proceeds sequentially according to the order in which rules for functional blocks appear in the database. A functional block may be composed of other lower level functional blocks. For recognition to proceed correctly, a block which is a component of another functional block should be defined before the latter. For this reason, the order in which rules for functional blocks appear in the database should be such that lower complexity blocks are defined before higher complexity blocks.

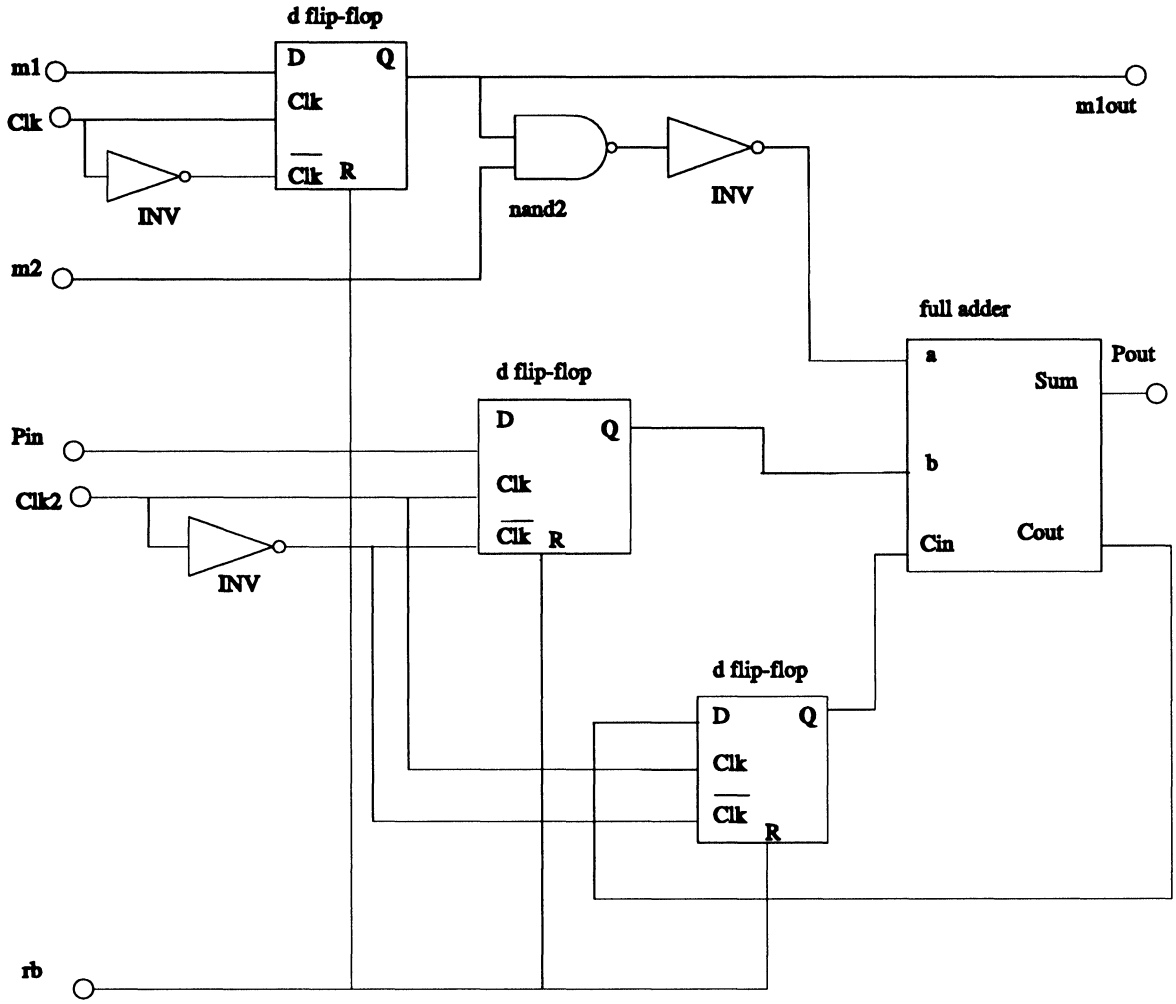

FIGURE 6 A Multiplier Circuit

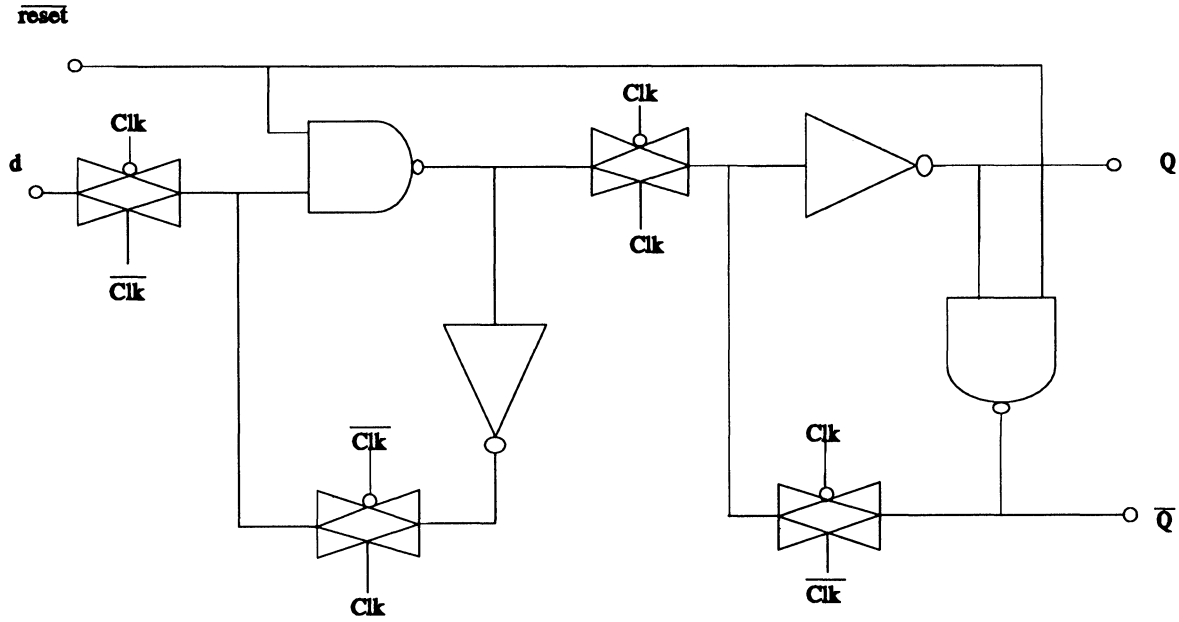

FIGURE 7 D Flip-Flop Circuit with Reset.

```
GATE:transmission
input al
output dfll
gate_inputs clklb clkl

GATE:transmission
input pin
output df21
gate_nputs clk2b clk2

GATE:transmission
input df13
output df11
gate-inputs clkl clklb

GATE:transmission
input df23
output df21
gate_inputs clk2 clk2b

GATE:transmission
input df33
output df31
gate_inputs clk2 clk2b

GATE:transmission
input cout
output df31
gate_nputs clk2b clk2

GATE:transmission
input df3qb
output df34
gate_inputs clk2b clk2

GATE:transmission
input df32
output df34
gate_inputs clk2 clk2b

GATE:transmission
input df2qb
output df24
gate_inputs clk2b clk2

GATE:transmission
input df22
output df24
gate_inputs clk2 clk2b

GATE:transmission
input df1qb
output df14
gate_inputs clk1b clk1

GATE:transmission
input df12
output df14
gate_inputs clkl clklb
```

```
Gate:inverter
input clkl
output clklb

GATE:inverter
input s2
output s3

GATE: inverter
input clk2
output clk2b

GATE:inverter
input df12
output df13

GATE: inverter
input df14
output slout

GATE:inverter
input df22
output df23

GATE:inverter
input df24
output s4

GATE:inverter
input df32
output df33

GATE:inverter
input df34
output s5

GATE: nand2
input [s5,s4]
output ad4

GATE: nand2
input [s5,s3]
output ad3

GATE: nand3
input [ad4,ad3,ad2]
output cout

GATE: nand2
input [s4,s3]
output ad2

GATE: and2_nor
input [x22,s5,adl]
output pout
```

```
Gate: and2-nor
input[x12,s4,s3]
output adl

GATE: nor2
input [adl,s5]
output x22

GATE: nor2
input [s3,s4]
output x12

GATE: nand2
input [rb,s5]
output df3qb

GATE: nand2
input [df31,rb]
output df32

GATE: nand2
input [rb,s4]
output df2qb

GATE: nand2
input [df21,rb]
output df22

GATE: nand2
input [s2, slout]
output s2

GATE: nand2
input {rb, slout}
output df1qb

GATE: nand2
input [df11, rb]
output df12

GATE: and2
input [df31, rb]
output df33

GATE: and2
input [df21,rb]
output df23

GATE: and2
input [n2, slout]
output s3

GATE: and2
input [df11,rb]
output df13
```

```
Function: exorgate
inputs x5 adl
output pout

*****
Function: exorgate
inputs s4 s3
output adl

*****
Function: exorgate
inputs adl s5
output pout

*****
Function:exorgate
inputs s3 s4
output adl

Function:full adder
inputs s4 s3 s5
sun pout
carryout cout

*****
Function:full adder
inputs s3 s4 s5
sun_pout
carryout cout
*****
Function: dffr
input cout
reset_input rb
outputs s5 df3qb
clock clk2
*****
Function: dffr
input pin
reset_input rb
outputs s4 df2qb
clock clk2
*****
Function: dffr
input n1
reset_input rb
outputs slout df1qb
clock clk1
*****
Function:nult cell
inputs n1 n2 pin rb
clocks clkl clk2
outputs slout pout
*****
```

FIGURE 8 Gates and Blocks Recognized in the Multiplier Circuit.

The order in which the input and output nodes are listed within the attributes for a functional block is important. In particular, if two or more of the inputs of a functional block are interchangeable, through backtracking, the program may report more than one instance of the same functional block with the same set of inputs but a different order. For example, the output generated for the multiplier circuit of Figure 6 shows two instances of the same full adder with the order of inputs changed.

RESULTS AND CONCLUSIONS

The block extracting system described in this paper translates a physical description of a CMOS circuit which is in terms of transistor interconnections, to a logic level description which is in terms of interconnections of logic functional blocks. The block extractor operates on a circuit to produce a list of gates and logic blocks in the circuit along with their input and output terminals.

The system is an aid in checking the physical design of a circuit against the intended logic level description. It facilitates the task of network comparison by translating it to a higher level in the circuit hierarchy. Rules describing the logic blocks used in a design should be present in the program database. If an instance of each logic block present in the original design is found in the circuit by the circuit recognition procedure, and the recognized blocks are interconnected correctly, the physical design can be assumed to be logically correct.

The system can also be used to check for the presence of a particular circuit block in the designed circuit. In this case, the structural description of the circuit block is provided and contains actual node names in the circuit. The program then checks for a specific circuit block with the given input and output terminals. If the node names in the description of the circuit block are left as variables, the program would check for all invocations of that block within the design.

Since the system extracts logic expressions at output nodes of blocks, it has a further application. It can be used to verify the logical behaviour of a cell which is composed of a single load-driver block or a pass block. In this case, the cell is described in terms of the boolean equations at its output nodes. Using the equivalence rule described earlier, the program can be made to check for an equivalence between the extracted boolean expressions and the reference expressions. Lastly, the system can be used to arrive at logical expressions at output nodes of blocks in the circuit.

The C preprocessing program consists of 200 lines of code. The block extracting program consists of about 2300 lines of Prolog code and is implemented in three stages. The first stage is the circuit partitioning stage, in which a CMOS circuit is partitioned into smaller subcircuits. The second stage consists of extracting logic expressions at output nodes of circuit blocks. In the third stage circuit recognition rules are used to identify logic blocks within a circuit. The program has been implemented in the dialect of Prolog known as the Edinburgh syntax, on the MV/10000 Data General computer running under the AOS-VS operating system. The described system is a research prototype. The ease of implementing an approach has overridden the need to achieve run-time efficiency.

Logic blocks in a number of circuits were recognized using the approach described. Sample results for different circuits are shown in Table I. These results show the program performance for a set of circuits whose size covers a moderately broad spectrum. Functional description of the sample circuits mentioned in Table I along with further results including those for additional circuits can be found in [21].

The program run time contains a factor which is dependent on the computer system used. In the absence of a Prolog compiler, the program runs on an interpreter. This could be a disadvantage with regard to the run time of the program. A more efficient implementation of this system would have the first two stages of the block extractor written in C. The third stage of the program involves circuit recognition based on rules. This stage makes use of the symbolic features of Prolog and would be difficult to carry out in a language such as Pascal or C.

The run time is also dependent on the size of the program database. As more rules for recognizing logic blocks are added to the database, the time taken to recognize a logic block increases. There is a restriction on the size of gates which can be recognized. This restriction arises from the fact that the number of input assignments to be considered in a gate is n!

TABLE I
Sample Run Times

Circuit	1	2	3	4
Number of transistors	9	22	108	2552
Number of nodes	8	14	61	1891
Number of gates and blocks extracted	4	9	47	1210
CPU time in seconds	7.4	22	134	3284

where n is the number of gate inputs. The recognition time is dependent not only on the size of the circuit, that is the number of transistors present within the circuit, but also on the complexity of gates and logic blocks in the circuit. A large circuit with a number of simple gates (having a small number of inputs) would take proportionally less time than a smaller circuit containing a few complex gates (having a large number of inputs). The time taken to recognize functional blocks also depends to some degree on the way the rules for functional blocks are framed. What is important to note is that the increase in run time does not increase exponentially as the size of the circuit increases.

EXTENSIONS TO THE WORK DONE

Possible extensions to the block extraction system are mentioned below.

1. Connectivity (Netlist) comparison at a higher level: If a reference description of the circuit is provided in the form of an interconnection of gates and logic blocks, it can be compared with the extracted list of gates and logic blocks to check for the correctness of the circuit. In order to do so, the block extraction program can be run on both, the netlist obtained from the layout through a circuit extractor and the netlist of the specified circuit which could have been entered using a schematic capture system. By pattern matching, the gates and logic blocks from the two netlists can be compared and checked for equivalence. The present recognition procedure reports the presence or absence of blocks within a circuit. It does not have the ability to report discrepancies between a reference circuit description and the extracted block level description.
2. Simulation at a higher level: The recognition of logic blocks could enable symbolic or logic simulation at the functional block level.
3. Extraction of higher level behaviour: Given an interconnection of logic blocks in a circuit, an attempt could be made to derive the behavioural description of the circuit. For this purpose, the behaviour of each functional block would be stored in a suitable format. The behaviour of the circuit would then be extracted from the behaviour of its components and a description of their interconnection.
4. Operational Speedup: The nature of the circuit and block extraction problem is such that this process lends itself to being a candidate for solution on a parallel processor computing system. The problem could be partitioned along geometric boundaries and the search for independent and coordinated solutions could be performed on distinct processors.

References

[1] S. Leinwand and T. Lamdan, "Design Verification Based on Functional Abstraction," *Proceedings of the 15th Design Automation Conference*, 1978, pp. 353–359.

[2] A. Gupta, "ACE—A Circuit Extractor," *Proceedings of the 20th Design Automation Conference*, 1983, pp. 721–725.

[3] W.S. Scott and J.K. Ousterhout, "Magic's Circuit Extractor," *IEEE Design and Test*, February 1986, pp. 24–34.

[4] M. Takashima, T. Mitsuhasti, T. Chiba, and K. Yoshida, "Programs for Verifying Circuit Connectivity of MOS/LSI Artwork," *Proceedings of the 19th Design Automation Conference*, 1982, pp. 544–550.

[5] T. Watanabe, M. Endo, and N. Miyahara, "A New Automatic Logic Interconnection Verification System for VLSI Design," *IEEE Transactions on Computer-Aided-Design of Integrated Circuits and Systems*, Vol. CAD-2, No. 2, April 1983, pp. 70–81.

[6] L. Scheffer and R. Apte, "LSI Design Verification using Topology Extraction," *Proceedings of the 16th Design Automation Conference*, 1979, pp. 149–153.

[7] P.J. Russell, "Algorithms for Generalized On-Chip FET Circuit Recognition," *IBM Technical Disclosure Manual*, Vol. 21, No. 2, July 1972, pp. 815–819.

[8] P.J. Russell, "Physical to Logical Checking of FET LSI Chips," *IBM Technical Disclosure Manual*, Vol. 21, No. 2, July 1972, pp. 822–824.

[9] F. Luellan, T. Hoepken, and E. Barke, "A Technology Independent Block Extraction Algorithm," *Proceedings of the 21st Design Automation Conference*, 1984, pp. 610–615.

[10] M. Boehner, "LOGEX-An Automatic Logic Extractor from Transistor to Gate Level for CMOS Technology," *Proceedings of the 25th Design Automation Conference*, 1988, pp. 517–522.

[11] R.E. Bryant, "MOSSIM: A Switch Level Simulator for MOS LSI," *Proceedings of the 18th Design Automation Conference*, 1981, pp. 786–790.

[12] I. Bratko, *Prolog Programming for Artificial Intelligence*, Addison-Wesley Publishing Company, 1986.

[13] N. Suzuki, "Concurrent Prolog as an Efficient VLSI Design Language," *Computer*, Vol. 18, No. 2, Feb. 1985, pp. 33–39.

[14] F. Maruyama and M. Fujita, "Hardware Verification," *Computer*, Vol. 18, No. 2, Feb. 1985, pp. 22–32.

[15] N. Srinivas and V.D. Agrawal, "Prove: Prolog Based Verifier," *IEEE International Conference on Computer-Aided-Design*, 1986, pp. 306–309.

[16] A.C. Papaspyridis, "A Prolog Based Connectivity Verification Tool," *Proceedings of the 25th Design Automation Conference*, 1988, pp. 523–527.

[17] W.S. Scott et al, "1986 VLSI Tools," Report No. UCB/CSD 86/272, December, 1985, Computer Science Division, University of California, Berkeley.

[18] V. Rao and T. Trick, "Network Partitioning and Ordering for MOS VLSI Circuits," *IEEE Transactions on Computer-Aided-Design*, Vol. CAD-6, No. 1, Jan. 1987, pp. 128–143.

[19] G.F. Pfister, "Algorithms for Deducing the Logical Behavior of Arbitrary FET Circuits," IBM Technical Disclosure Bulletin, Vol. 27, No. 2, July 1984, pp. 1168–1179.

[20] G. Ditlow, W. Donath and A. Ruehli, "Logic Equations for MESFET Circuits," *IEEE International Symposium on Circuits and Systems*, 1983, pp. 752–755.

[21] I. Bhasin, "Recognition of Logic Blocks in CMOS Circuits," *M.S.E.E. Thesis,* Department of Electrical Engineering, Virginia Polytechnic Institute and State University, 1988.

Biographies

INDERPREET BHASIN was born in Nainital, India in 1963. He received the B. Tech. degree in Electrical Engineering from the Indian Institute of Technology, Bombay, in 1986, and the M.S. degree in Electrical Engineering from Virginia Polytechnic Institute in 1988. Since 1989 he has been a design engineer with Intel Corporation in Portland, Oregon, where he works on the design of microprocessor chips. His areas of interest include logic and circuit design and design verification of VLSI devices.

DR. JOSEPH G. TRONT is an Associate Professor of Electrical Engineering at Virginia Polytechnic Institute and State University where he teaches both graduate and undergraduate courses in computer engineering and electronics. His research interests include VLSI design, fault-tolerant computing, digital circuit modeling and simulation, VLSI testing, and microprocessor applications. His work in VLSI has been on the development of CAD tools as well as the development of schemes for implementing fault-tolerant architectures as VLSI circuits. He has worked on modeling single-event upset effects in integrated circuits. He has also been involved in the design and modeling of parallel computer architectures for solving combinatorial problems.

Dr. Tront received a B.E.E. in 1972 and an M.S.E.E. in 1973 from the University of Dayton. He received the Ph.D. degree from the State University of New York at Buffalo in 1978. He joined the faculty at Virginia Tech in 1978.