Design Procedures for Differential Cascode Voltage Switch Circuits

KAN M. CHU AND DAVID I. PULFREY, MEMBER, IEEE

Abstract — Differential cascode voltage switch (DCVS) logic is a CMOS circuit technique which has potential advantages over conventional NAND/NOR logic in terms of circuit delay, layout density, power dissipation, and logic flexibility. In this paper, two procedures are presented for constructing DCVS trees to perform random logic functions. The first procedure uses a Karnaugh mapping technique and is a very powerful pictorial method for hand-processing designs involving up to six variables. The second procedure is a tabular method based on the Quine-McCluskey approach and is suitable for functions with more than six variables. Both of these procedures are considerably easier to implement than a recently proposed algebraic technique which relies upon decomposition and factorization of Boolean expressions. Several DCVS circuits that have been synthesized by the Karnaugh map (K-map) and tabular procedures are presented.

I. INTRODUCTION

THE implementation of CMOS random logic design with differential cascode voltage switch (DCVS) logic has many advantages over the traditional NAND/NOR logic approach. The most obvious advantage is in device count; it appears that DCVS circuits will usually require fewer transistors, of both n- and p-type, than the two-level NAND/NOR implementation of random logic [1]. A recent example of this is a leverage circuit in which a large number of delay stages have been compressed into a single CVS gate [2]. Although this CVS circuit has an increased stack height (i.e., more transistors in each cascode chain), the reduction in the number of stages results in the further advantages of a shorter circuit delay, a smaller chip area, and less power dissipation. Another general advantage of DCVS is the increase of logic flexibility that is afforded, especially in those instances where some complex function must be implemented in domino CMOS. Standard domino logic suffers from the fact that inverting logic gates cannot be implemented. However, clocked DCVS provides complementary outputs and therefore overcomes this restriction [1].

All the above advantages indicate that DCVS logic represents an important new direction in CMOS logic design. The only existing procedure for the design of DCVS trees is an algebraic technique based on the identification of subexpressions common to two or more Boolean functions [3]. The decomposition and factorization techniques involved in this approach are quite mathematical. As such, the method does not provide the insight into circuit behavior which is often important for IC designers.

This paper introduces two other, much simpler and more practical methods for constructing DCVS trees. The first procedure utilizes the pictorial nature of the Karnaugh map (K-map). This hand-processing method is shown to be an efficient approach to realizing low device-count circuits for functions of up to five or six variables. However, the complexity of K-maps suddenly increases when more than five variables are considered. Accordingly, a second procedure which has a uniform procedural complexity for \( n \) variables has been developed. The method is tabular in nature and is a modified form of the Quine-McCluskey method [4].

Note that a unique one-to-one correspondence between a Boolean expression and a DCVS tree structure does not exist [5]. Thus the above design procedures can produce several tree structures to realize a particular logic operation. Also, for a given structure, some of the input variables may be allowed to permutate.

The two DCVS design procedures proposed here can be used to implement any Boolean function provided the appropriate truth tables are known. Examples we have investigated in CMOS designs include adder cells, magnitude comparators, and multiplier circuits [6].

II. THE DIFFERENTIAL CASCODE VOLTAGE SWITCH TREE

Differential cascode switch circuits usually consist of a push–pull load and a pair of interrelated binary decision trees (or DCVS trees), as shown in Fig. 1. The DCVS tree is properly designed such that: 1) when the input vector \( x = (x_1, \ldots, x_n) \) is the true vector of the switching function \( Q(x) \), node \( Q \) is disconnected from ground and node \( Q \) is connected to ground by a unique conducting path through the tree; and 2) when \( x = (x_1, \ldots, x_n) \) is the false vector of \( Q(x) \), the reverse holds. A simple example is a two-way EXCLUSIVE-OR DCVS gate shown in Fig. 2(a). The functionality of this circuit...
positive that the DCVS tree can be constructed easily by intuition, especially for those kinds of Boolean functions with a recursive nature. For instance, a three-way XOR tree (Fig. 2(b)) can be built by replacing the $\overline{x_2}, x_2$ pair in Fig. 2(a) with another two-way XOR tree. Fig. 2(c) shows a general structure for an $n$-way XOR tree, with a stacking height equal to $n$.

Another interesting example of Boolean functions with recursive nature arises in the carry lookahead circuit [7], which can be implemented as a DCVS circuit with minimal design effort.

### III. Design of DCVS Trees Using the Karnaugh Map

#### A. Notations and Definitions

The input variable of a DCVS tree is represented by $x_i$, for $i=1,2,\cdots,n$. A literal is a variable $x_i$ or its negation $\overline{x_i}$. A cube is a set $P$ of literals such that $x_j = P$ implies $\overline{x_j} \notin P$.

In a Karnaugh map of $n$ variables, there are $2^n$ cells of which each represents a cube consisting of exactly $n$ literals. Cells that contain one's are called 1-cells (similarly, 0-cells). A 1-loop that encircles two adjacent 1-cells expresses a cube with one less literal than each of the cubes representing the original 1-cell (similarly, 0-loop). Suppose that two rectangular 1-loops, each consisting of $2^k$ 1-cells, are adjacent on a $K$-map. If these 1-loops express cubes, say $C_{x^k}$ and $C_{\overline{x^k}}$, we get a new rectangular 1-loop consisting of $2^{k+1}$ 1-cells by combining the two 1-loops, and the new 1-loop expresses cube $C$ (similarly for the 0-loops).

Before introducing the K-map algorithm, we give an example to demonstrate some of the ideas, i.e., given the Boolean function $Q = x_1x_2 + x_2x_3 + x_3x_1$ (which is the form of the carry-out function of a full adder), construct the corresponding DCVS tree. The K-map is shown in Fig. 3(a). The 1- and 0-loops are encircled properly to form the minimal cover for the 1- and 0-cells, respectively.

Fig. 3(b) illustrates the resulting DCVS tree pair. The tree attached to node $\overline{Q}$ is derived from the 1-cells and is called the 1-tree. Similarly, the 0-tree is derived from the 0-cells and is attached to node $Q$. Note that the 1- and
0-trees are disjointed because the 1-cells and 0-cells have been grouped separately. This DCVS circuit requires ten $N$ devices to realize the function $Q$.

The K-map procedure does more than just construct the two disjointed 1- and 0-trees. It also allows the maximum commonality between these two trees to be explored; from this a “shared” tree structure leading to the minimization of device count can be developed.

Suppose a 1-cell (0-cell) representing the cube $X_1P$ and a 0-cell (1 cell) representing the cube $X_1P$ simultaneously exist. Then the cell corresponding to the cube $P$ is defined as a 10-cell (01-cell). These 01- or 10-cells act as individual cells of two different types. A 01-loop (10-loop) can be formed by encircling two or more adjacent 01-cells (10-cells).

With these concepts added, we revisit the previous example. The K-map shown in Fig. 4(a) has three types of encirclements, namely, 0-, 1-, and 10-loop. The “shared” tree corresponding to the 10-loops is first constructed (Fig. 4(b)), and then more branches corresponding to the 1- and 0-loop are added to form a complete DCVS tree (Fig. 4(c)). Note that only eight $N$ devices are required, which is two devices fewer than for the disjointed tree in Fig. 3(b). However, the number of stacked levels has increased to three.

0-trees are disjointed because the 1-cells and 0-cells have been grouped separately. This DCVS circuit requires ten $N$ devices to realize the function $Q$.

The K-map procedure does more than just construct the two disjointed 1- and 0-trees. It also allows the maximum commonality between these two trees to be explored; from this a “shared” tree structure leading to the minimization of device count can be developed.

Suppose a 1-cell (0-cell) representing the cube $X_1P$ and a 0-cell (1 cell) representing the cube $X_1P$ simultaneously exist. Then the cell corresponding to the cube $P$ is defined as a 10-cell (01-cell). These 01- or 10-cells act as individual cells of two different types. A 01-loop (10-loop) can be formed by encircling two or more adjacent 01-cells (10-cells).

With these concepts added, we revisit the previous example. The K-map shown in Fig. 4(a) has three types of encirclements, namely, 0-, 1-, and 10-loop. The “shared” tree corresponding to the 10-loops is first constructed (Fig. 4(b)), and then more branches corresponding to the 1- and 0-loop are added to form a complete DCVS tree (Fig. 4(c)). Note that only eight $N$ devices are required, which is two devices fewer than for the disjointed tree in Fig. 3(b). However, the number of stacked levels has increased to three.

0-trees are disjointed because the 1-cells and 0-cells have been grouped separately. This DCVS circuit requires ten $N$ devices to realize the function $Q$.

The K-map procedure does more than just construct the two disjointed 1- and 0-trees. It also allows the maximum commonality between these two trees to be explored; from this a “shared” tree structure leading to the minimization of device count can be developed.

Suppose a 1-cell (0-cell) representing the cube $X_1P$ and a 0-cell (1 cell) representing the cube $X_1P$ simultaneously exist. Then the cell corresponding to the cube $P$ is defined as a 10-cell (01-cell). These 01- or 10-cells act as individual cells of two different types. A 01-loop (10-loop) can be formed by encircling two or more adjacent 01-cells (10-cells).

With these concepts added, we revisit the previous example. The K-map shown in Fig. 4(a) has three types of encirclements, namely, 0-, 1-, and 10-loop. The “shared” tree corresponding to the 10-loops is first constructed (Fig. 4(b)), and then more branches corresponding to the 1- and 0-loop are added to form a complete DCVS tree (Fig. 4(c)). Note that only eight $N$ devices are required, which is two devices fewer than for the disjointed tree in Fig. 3(b). However, the number of stacked levels has increased to three.

Generally, the reduction of the number of devices by tree sharing does not necessarily cause an increase in the number of stacked levels. In fact, the heuristic procedures that will be discussed tend to optimize both device count and number of stacked levels.

B. K-Map Procedure

The procedure consists of four steps.

1) Identify four different types of cells in the K-map, namely, 0-, 1-, 01-, and 10-cells.

2) Find a minimal cover for all the 01-cells. Construct the tree corresponding to this minimal cover. The variable $x_1$'s in each of the tree branches are arranged from top to bottom in ascending order with magnitude of $i$. Always construct tree branches corresponding to loops of smaller size first. The top pair of control inputs are $x_1$ associated with node $Q$, and $\bar{x}_1$ associated with node $\bar{Q}$. The sources of the transistors with gates $x_1$ and $\bar{x}_1$ are connected together.

3) From the prime implicants of all the 10-cells, find a minimal cover such that the tree constructed may share
some of the branches with the tree in step 2. Contrary to
step 2, the top pair of control inputs are $x_1$ associated with
node $Q$, and $x_1$ associated with node $\bar{Q}$.

4) Find a minimal cover for the remaining 0-cells and
1-cells. While constructing the tree, always look for the
sharing of tree branches. The root of the 0-tree (1-tree) is
connected to node $Q$ (node $\bar{Q}$).

This procedure may create different tree structures if
$x_i$'s are permuted (e.g., $x_1$ and $x_2$ variables are in-
terchanged). Also, there may be several ways to choose a
minimal cover, and to share tree branches.

As an example, given a four-variable K-map as shown in
Fig. 5(a), application of steps 1 and 2 generates the tree
structure in Fig. 5(b). Further, applying step 3 generates
the complete DCVS tree in Fig. 5(c). Step 4 has been
skipped because there are no remaining 0-cells and 1-cells.

A different way of encircling the K-map, as shown in
Fig. 6(a), leads to a different tree structure; see Fig. 6(b).
Note that the 10-cells are not covered minimally in this
manifestation, and thus the stack level in some of the tree
branches is increased. This undesirable feature, combined
with the large parasitic capacitances associated with the
numerous shared source and drain connections, indicates
that the circuit of Fig. 5(c) would have superior electrical
performance to that of the circuit in Fig. 6(b).

IV. DESIGN OF DCVS TREES BY A TABULAR METHOD

The tabular method described here makes use of the
Quine–McCluskey method [4] of finding prime implicants
and their minimal covering set. A list conventionally con-
sists of two fields, namely, the input vector ($x_1, \cdots, x_n$) on
the right and its decimal representation on the left (see
Table I). The input vectors are grouped into records in an
ascending order of their index (number of ONE's in their
binary representation). We start with a 1-list (list contain-
ing ONE's of the function), and a 0-list (list containing
ZERO's). From these two lists, we generate another two
lists, namely, a 10-list and a 01-list. The mechanism is
analogous to the generation of 10-cells and 01-cells from
the 1-cells and 0-cells in the K-map approach. Selection of
minimal covers from the 0-list, 1-list, 10-list, and 01-list
using a modified Quine–McCluskey procedure results in a
DCVS tree structure shown in Fig. 7.

A. Tabular Procedure

1) Draw a 1-list which contains all the true vectors
($x_1, \cdots, x_n$) of the function $Q$. The list is subdivided into
records with increasing index $i$ from top to bottom. Similarly,
a 0-list which contains all the false vectors of $Q$ is
drawn. The record with index $i$ is notated as record $i$.

2) For $i = 1$ to $n$ : In the 1-list each row starting with
$x_1 = 1$ within record $i$ is compared with the rows within
record $i-1$ of the 0-list. If the reduced vector ($x_2, \cdots, x_n$)
of the two rows is the same, then check these two rows in
the 1-list and 0-list, respectively, and add a row entry to
the 10-list. The format of the 10-list is slightly different
and is shown in Table II. The variable $x_1$ is no longer
required.

3) For $i = 0$ to $n$ : In the 1-list, each row starting with
$x_1 = 0$ within record $i$ is compared with the rows within
record $i+1$ of the 0-list. Similarly, if the reduced vector
($x_2, \cdots, x_n$) of the two rows is the same, check

![Fig. 6](image_url)

![Fig. 7](image_url)
TABLE II
TYPICAL FORMAT FOR A 10-LIST

<table>
<thead>
<tr>
<th>Decimal representation</th>
<th>Reduced input vector</th>
<th>Record 1</th>
<th>Record 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE III
The 10-List of a 3-Bit Magnitude Comparator

<table>
<thead>
<tr>
<th>Decimal representation</th>
<th>Reduced input vector</th>
<th>Input vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>4</td>
<td>0 0 1 0 0</td>
<td>0 0 1 0 0</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 0 0</td>
<td>0 0 1 0 0</td>
</tr>
<tr>
<td>5</td>
<td>0 0 1 1 0</td>
<td>0 0 1 1 0</td>
</tr>
<tr>
<td>6</td>
<td>0 0 1 1 0</td>
<td>0 0 1 1 0</td>
</tr>
<tr>
<td>12</td>
<td>0 0 0 0 1</td>
<td>0 0 0 0 1</td>
</tr>
<tr>
<td>24</td>
<td>1 1 0 0 0</td>
<td>1 1 0 0 0</td>
</tr>
<tr>
<td>18</td>
<td>1 0 0 1 0</td>
<td>1 0 0 1 0</td>
</tr>
<tr>
<td>7</td>
<td>0 0 0 1 0</td>
<td>0 0 0 1 0</td>
</tr>
<tr>
<td>13</td>
<td>0 1 1 0 1</td>
<td>0 1 1 0 1</td>
</tr>
<tr>
<td>25</td>
<td>1 1 0 0 1</td>
<td>1 1 0 0 1</td>
</tr>
<tr>
<td>26</td>
<td>1 1 1 0 0</td>
<td>1 1 1 0 0</td>
</tr>
<tr>
<td>15</td>
<td>0 1 1 1 1</td>
<td>0 1 1 1 1</td>
</tr>
<tr>
<td>27</td>
<td>1 1 0 1 1</td>
<td>1 1 0 1 1</td>
</tr>
<tr>
<td>30</td>
<td>1 1 1 1 0</td>
<td>1 1 1 1 0</td>
</tr>
</tbody>
</table>

TABLE IV
The Prime Implicant Table of the 10-List of Table III

<table>
<thead>
<tr>
<th>Prime implicants</th>
<th>Decimal representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>xy</td>
<td>0 1 3 4 5 6 7 12 13 15</td>
</tr>
<tr>
<td>xy</td>
<td>18 24 25 26 27 28 30</td>
</tr>
</tbody>
</table>

these two rows and add a row entry to the 01-list. The format of the 01-list is the same as that of the 10-list.

4) Apply the Quine–McCluskey method of finding prime implicants to the rows in the 10-list and the 01-list. Select a minimal covering set for each of the two lists by row and column dominance procedures, and look for a maximum amount of sharing of tree branches when constructing the corresponding trees. Thus a “shared” tree is built.

5) Apply the conventional procedure of selecting a minimal covering set for the unchecked rows in the 0-list and 1-list. Construct the trees corresponding to these two minimal sums, by adding more branches to the “shared” tree. Thus a DCVS tree structure of the form shown in Fig. 7 is completed.

As an example, consider the design of a 3-bit magnitude comparator by the tabular method. The circuit compares two binary numbers, \( A = A_3 A_2 A_1 \) and \( B = B_3 B_2 B_1 \), and the output \( Q = 1 \) whenever \( A > B \). We assign the variables \( (x_1, x_2, x_3, x_4, x_5, x_6) \) equal to \( (A_3, B_3, A_2, B_2, A_1, B_1) \). Note that a different assignment will lead to a different tree structure.

By step 1 of the procedure, we readily tabulate the 1-list (totaling 28 rows), and the 0-list (36 rows). After step 2 is performed, a 10-list as shown in Table III is drawn. Application of step 3 indicates that no 01-list can be

TABLE V
The 1-List and Its Minimal Sum for the Magnitude Comparator

<table>
<thead>
<tr>
<th>Decimal representation</th>
<th>Input vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>8</td>
<td>0 0 1 0 0</td>
</tr>
<tr>
<td>9</td>
<td>0 1 0 0 0</td>
</tr>
<tr>
<td>10</td>
<td>0 0 1 0 1</td>
</tr>
<tr>
<td>40</td>
<td>1 0 1 0 0</td>
</tr>
<tr>
<td>34</td>
<td>1 0 0 0 1</td>
</tr>
<tr>
<td>11</td>
<td>0 1 0 0 0</td>
</tr>
<tr>
<td>14</td>
<td>0 1 1 1 1</td>
</tr>
<tr>
<td>41</td>
<td>1 0 1 0 0</td>
</tr>
<tr>
<td>42</td>
<td>1 0 1 0 1</td>
</tr>
<tr>
<td>43</td>
<td>1 0 1 0 1</td>
</tr>
<tr>
<td>46</td>
<td>1 0 1 1 1</td>
</tr>
</tbody>
</table>

\[ \text{minimal sum} = x_1 x_2 x_3 x_4 x_5 + x_2 x_3 x_4 x_5 + x_2 x_3 x_4 x_5 + x_2 x_3 x_4 x_5 \]

Fig. 8. The shared DCVS tree circuit corresponding to the 10-list of Table IV.

Fig. 9. The complete DCVS tree for the 3-bit magnitude comparator.
generated. By step 4, a prime implicant table as shown in Table IV is derived from the 10-list, and these prime implicants actually form a minimal covering set. The “shared” tree is illustrated in Fig. 8. By step 5, the unchecked rows of the 1-list and 0-list result in Tables V and VI, respectively. Their corresponding minimal sums worked out by the Quine–McCluskey method are also indicated. The complete DCVS tree is illustrated in Fig. 9.

V. CONCLUSION

Two methods have been presented for designing DCVS trees. The first one is a pictorial method based on the Karnaugh map, and is very powerful for designing by hand trees with up to six input variables. The second one is a tabular method making use of the Quine–McCluskey procedure. In each case, circuits with minimal transistor count can be realized in a very straightforward manner.

For a DCVS circuit with more than six stacked levels, the performance may be reduced because of charging and discharging the parasitic drain and source capacitances through long chains of transistors. So, for circuits requiring high speed, it may be better to break up complicated logic into DCVS circuits of six or less variables. In such cases, the K-map design procedure may prove particularly useful.

REFERENCES


Kan M. Chu was born in Hong Kong on May 16, 1962. He received the B.Eng. (Honors E.E.) degree from McGill University, Montreal, Canada, in 1984 and the M.A.Sc. (E.E.) degree from the University of British Columbia, Vancouver, B.C., Canada, in 1986. He plans to work towards a Ph.D. degree in electrical engineering. His research interests are in CMOS integrated circuit design and semiconductor device modeling.

David L. Pulfrey (M’73) is a Professor in the Electrical Engineering Department at the University of British Columbia, Vancouver, B.C., Canada. His research interests are in the fields of semiconductor device physics and integrated-circuit design. He has worked on the topics of electrical breakdown in thin dielectrics, the preparation and properties of plasma-anodized thin-oxide films, and the analysis and fabrication of solar cell structures suited to large-area terrestrial applications. His present work at the University of British Columbia is in the areas of high-gain polysilicon emitter transistors, the characterization and applications of MIS tunnel junctions, and the algorithmic generation of IC macrocells. He is also associated with Microtel Pacific Research, Vancouver, B.C., Canada, engaged in the design of integrated circuits.