Computation of Disjoint Cube Representations Using a Maximal Binate Variable Heuristic[*]

Lokesh Shivakumaraiah and Mitchell A. Thornton
Department of Electrical and Computer Engineering
Mississippi State University
{ls1,mitch}@ece.msstate.edu

ABSTRACT

A method for computing the Disjoint-Sum-Of-Products (DSOP) form of Boolean functions is described. The algorithm exploits the property of the most binate variable in a set of cubes to compute a DSOP form. The technique uses a minimized Sum-Of-Products (SOP) cube list as input. Experimental results comparing the size of the DSOP cube list produced by this algorithm and those produced by other methods demonstrate the efficiency of this technique and show that superior results occur in many cases for a set of benchmark functions.

Index Terms - *Binate Function Variables, Cube List Representations Disjoint-sum-of-products, Unate Function Variables*

I. Introduction

Representing a Boolean function in the form of a *Disjoint-Sum-Of-Products* (DSOP) cube list has many advantages in *Computer Aided Design* (CAD) tools. DSOPs can be used to efficiently compute the spectra of Boolean functions [3,4,8], in the minimization of *Exclusive-OR-Sum-of-Products* (ESOPs) [5,9,10,11] and to quickly find the complement of Boolean functions [6]. This fact has encouraged researchers to invent new algorithms [1,2] to generate the DSOP forms of Boolean functions efficiently.

Given a logic function in the form of a minimized *Sum-Of-Products* (SOP) cube list, the algorithm presented here computes the corresponding DSOP form by splitting the SOP cube list according to the variable that occurs most often in both the complemented and uncomplemented form. The two resulting cube lists represent Shannon cofactor functions about the splitting variable. The motivating paradigm of this technique is to attempt to produce cofactor cube lists that contain roughly an equal number of supporting minterms.

Choosing the variable that exists in complemented and uncomplemented forms equally in a minimized SOP cube list corresponds to choosing the "most binate" variable of the subfunction. By choosing such a variable about which to decompose the function is likely to yield cofactor functions of approximately equally complex functions in terms of supporting minterm counts.

[*] This research was supported in part by National Science Foundation grants CCR-0000891 and CCR-0097246.

Section 2 gives the definitions of various terms used in the paper. Section 3 describes the methodology used to compute the DSOP. Section 4 contains experimental results and, finally Section 5 gives the conclusions and ideas for future refinements and applications of these results.

II. Definitions

Definition 1: Boolean expressions: *A Boolean expression is an algebraic clause representing a relationship among a set of Boolean valued literals. A Boolean function can be represented by an equation containing Boolean expressions.*
e.g.: $f = x' \bullet y' \bullet z' + z$

Definition 2: Product/Cube: *A Boolean expression containing a set of literals conjuncted together (i.e. ANDed).*
e.g.: $f = x' \bullet y' \bullet z'$

Definition 3: Disjunctive Sum: *The inclusive OR function in Boolean algebra.*
e.g.: $f = x' + z$

Definition 4: Sum-of-products (SOP): *Two or more AND functions are ORed together to form a Sum-of-products expression. In this form, the product terms may or may not cover a common minterm. Figure 1 represents an SOP form of an example Boolean function using a Karnaugh map.*
e.g. $f = w \bullet y' + x + y \bullet z$

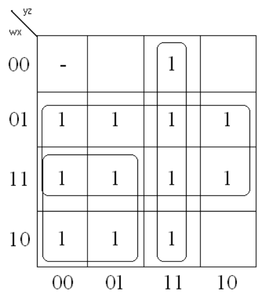

Figure 1. Example for Sum-of-products

Definition 5: Disjoint-Sum-of-Products (DSOP): *If no two product terms cover a common minterm, they are called a disjoint-sum-of-products. Figure 2 represents a DSOP form of the example Boolean function using a Karnaugh map.*

e.g. f = w•y’ + w•x’•y’ + x•y•z’ + y•z

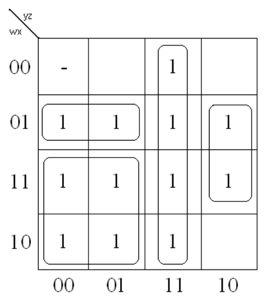

Figure 2. Example for Disjoint-sum-of-products

Definition 6: Unate functions: *A function is “ **Unate**” if it can be represented in simplest SOP form with each literal being complemented **or** uncomplented, but **not** both.*

Definition 7: Binate functions: *A function is “**Binate**” if it can be represented in simplest SOP form with each literal being present in both complemented **and** uncomplemented form.*

Examples of Unate, Binate functions are shown below.

f (x,y,z) = xy + yz	-Unate funtion
g(x,y,z) = x’y + y’z	-Unate in variables X and Z
	-Binate in variable Y

Figure 3. Unate and Binate functions.

Definition 8: Disjoint cubes: *Two cubes are said to be disjoint if their intersection of the set of minterms is null. The intersection is the operation of conjunction (i.e. the Boolean AND operation).*

For example, consider the two functions shown below.

f(w,x,y,z) = w’x
g(w,x,y,z) = x’z

The two cubes ‘f’ and ‘g’ are disjoint since the intersection of ‘f’ and ‘g’ is null.

Definition 9: Unate Variable: *A Boolean function is said to be unate in a variable if and only if the variable is present in complemented or uncomplemented form, but not both, in a minimized SOP form of the function.*

Definition 10: Binate Variable: *A Boolean function is said to be binate in a variable if and only if the variable is present in both complemented and uncomplemented form in a minimized SOP form of the function.*

III. Methodology

The algorithm presented here generates the DSOP form of a function given an SOP cube list representing it. The cube lists are read from netlist files in a **`.pla`** format [7]. This netlist represents a function as a set of cubes that cover all minterms of the function. This algorithm initially reduces the number of covering cubes by running *espresso* [7] and then splitting the cubes into two lists representing Shannon cofactors about the apparently most binate variable. Thus, the algorithm proceeds by building a binary tree of buffers containing cofactor cube lists. The algorithm is described in pseudocode form as shown in Figure 4.

DSOP Cube List Generation Algorithm:

```
step I: Read the pla file, run ESPRESSO

step II: Map input variables from Boolean domain to
           integer domain
         Boolean ‘0’ → Integer ‘+1’
         Boolean ‘1’ → Integer ‘-1’

step III: Compute column sums

step IV: Find the most binate variable
   case 1: If the column sums include zero
         a.  if one of the column sum is zero
                 binate variable => variable whose column
             sum equals to zero
         b.  if more than one column sum is zero
                 binate variable => any of the variable
             whose column sum equals to zero

   case 2: If the column sums does not include zero

         a.  if there is no tie in the column sums
               binate variable => variable whose magnitude
                                  of the column sum is close
                                    to zero
         b.  if there is no tie in the column sums
               binate variable => any of the variable whose
                                  column sum equals to
                                  zero

step V: Divide the cubes according to the most binate
variable

   if ( binate variable of the cube == 0 )
      then write the cube to buffer A
   else if ( binate variable of the cube == 1 )
      then write the cube to buffer B
   else if ( binate variable of the cube == - )
```

replace the don’t care with ‘0’ and write to buffer A
replace the don’t care with ‘1’ and write to buffer B

step VI: Repeat steps I, II, III, IV, V for files A and B, recurse until single cubes are obtained.

step VII: Write the single cubes obtained to a separate output file.

Figure 4. Algorithm

The output file thus contains an array of DSOP cubes. Figure 5 illustrates the flow of the algorithm.

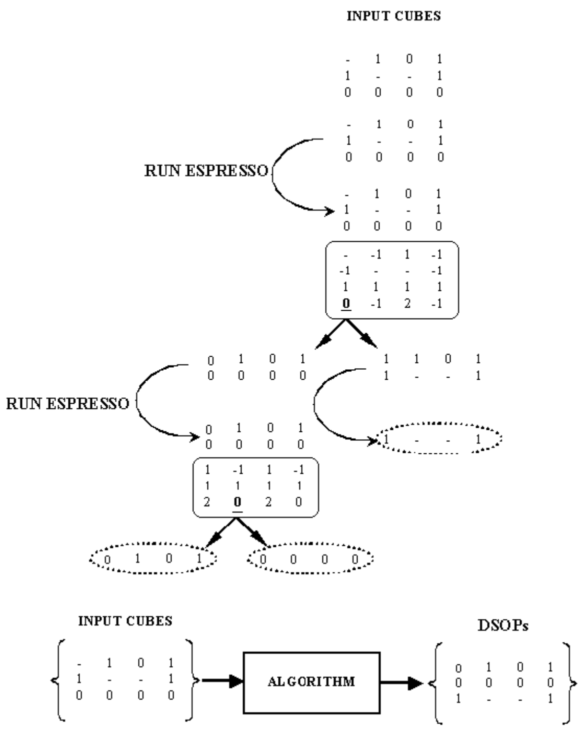

Figure 5. Steps of execution

The number of buffers at each level and the buffer operations is shown in Figure 5. At a given point of time only the buffers in the kth level and in the (k+1)th level are present (shown as the solid boxes in Figure 5) and the other buffers will be deleted as soon as possible to preserve memory. At any given instance of time only three buffers (shown inside the circle in Figure 5) and an output buffer being written to collects the DSOP cubes will be used. Among the buffers present inside the circle, one buffer in the kth level is being read from and two buffers in the (k+1)th are written into. Thus, there will be a sweeping operation of the buffers one at a time in the kth level to create two new buffers in the (k+1)th level. Figure 6 shows the creation of new buffers in the (k+1)th level from the kth level buffer.

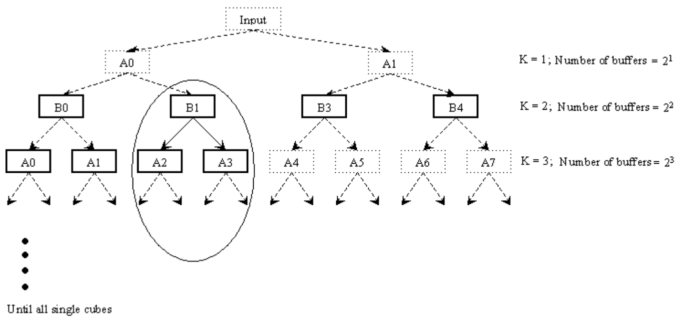
Input
A0
A1
K = 1; Number of buffers = 2¹
B0
B1
B3
B4
K = 2; Number of buffers = 2²
A0
A1
A2
A3
A4
A5
A6
A7
K = 3; Number of buffers = 2³
Until all single cubes

Figure 6. Dynamic Creation of Buffers

The algorithm is implemented in the ‘C’ programming language and its pseudocode presentation is given below. A legend containing the identifiers and their meaning as used in the pseudocode is given below and the pseudocode is explained in Figure 7.

Identifier Legend:

k	:	represents the kth level
in.pla	:	input pla
A_j.pla	:	jth buffer generated at kth level
B_j.pla	:	jth buffer generated at (2 * k)th stage
S_i	:	represents the column sum of the ith column
Aflag	:	Aflag = 1, represent that buffers of kth level are present
Bflag	:	Bflag = 1, represent that buffers of (2 * k)th level are present
binVar	:	position of the most binate Variable
incols	:	number of input variables
outcols	:	number of input variables
rows	:	number of product terms/cubes
array[ ][ ]	:	array used to compute the column sums
min	:	variable to store the minimum column sum
notfound	:	flag to count the number of buffers not present

Algorithm Pseudocode:

```
step 1: initialize

        Aflag ← 0
        Bflag ← 1

step 2: read the input.pla file
        run espresso on the input file
         store them in an array of size rows * incols
        min ← S[ 0 ]
        for ( i = 0; i < inputcols; i++ )
              S[ i ] ← 0 ;


step 3: mapping
        for each variable in the cube set do
        if Boolean ‘0’ → Integer ‘+1’
        if Boolean ‘1’ → Integer ‘-1’

step 4: Compute column sums

    for(m =0; m < rows; m++ )
        for(j =0; i < incols; i++)
            S[m] ← S[ m ] + array[ m ][ j ] ;

step 5: find the most binate variable

    for (i = 0; i < incols ; i ++ )
        if ( S[ i ] == 0 )
                binVar ← i ;
                break;
        else  if ( | S[ i ] | <  min )
                    min ← | S [ i ] |;
                binVar ← i ;

step 6: split the cubes according to the most binate variable

    if (array [ i ][ binVar ] == ‘0’)
        write the cube to buffer A0.pla ;
     else if (array [ i ][ binVar ] == ‘0’)
        write the cube to buffer A1.pla ;
    else if (array [ i ][ binVar ] == ‘-’)
    {
      write the cube to buffer A0.pla
        with array[ i ][binVar] ← ‘0’;
      write the cube to buffer A1.pla
        with array[ i ][binVar] ← ‘1’;
    }
    if ( singlecube )
        write to outputfile;
        exit;

    Aflag ← 1 ;



Step 7: loop until you get single cubes

    for ( j = 0 ;       ;  j++ )
        for( i = 0 ; i < pow (2, j) ; i ++)
            notfound ← 0 ;
            if(Aflag == 1 )
                input.pla ← Ai.pla ;
            else if(Bflag == 1 )
                input.pla ← Bi.pla ;

            if (input.pla == NULL)
                notfound ← notfound++ ;

            if ( notfound == 0 )
                if (single cube)
                    do step 2;
                    write the cube to the output.pla
                    continue;
                else
                    do step 2;
                    do step 3;
                    do step 4;
                    do step 5;
                    if (array [ i ][ binVar ] == ‘0’)
                        if (Bflag == 1 )
                        write the cube to buffer A(2 * i).pla ;
                            if (Aflag == 1 )
                        write the cube to buffer B(2 * i).pla ;
                else if (array [ i ][ binVar ] == ‘0’ )
                        if (Bflag == 1 )
                        write the cube to buffer A(2 * i  + 1).pla ;
                            if (Aflag == 1 )
                        write the cube to buffer B(2 * i  + 1).pla ;
                else if (array [ i ][ binVar ] == ‘-’)
                        if (Bflag == 1 )
                        write the cube to buffer A(2 *i).pla  with
                        array[ i ][binVar] ← ‘0’;
                        write  the  cube  to  buffer  A(2 *i   + 1).pla
with
                        array[ i ][binVar] ← ‘1’;

            if (notfound == pow( 2,j ) )
                exit;
            end of ‘i’ loop
        complement Aflag and Bflag;
        end of ‘j’ loop
```

Figure 7. DSOP Pseudocode

IV. Experimental Results

Table I reports the results of the algorithm described here and compares them with other methods. The table contains the name of the benchmark **`.pla`** file in the first column. The second column gives the size of cube list (in terms of the number of product terms) after minimizing it using *espresso*. The third column gives the number of product terms after finding the DSOP cube list using the new algorithm. The fourth column gives the number of DSOPs produced using *espresso* algorithm with the **`–Ddisjoint`** flag. The **`–Ddisjoint`** flag instructs *espresso* to heuristically determine a DSOP form instead of the normal reduced SOP form. The fifth column contains the results of a program (DJ) that generates a DSOP form that is described in [9] and was originally developed as a preprocessor for the MINT ESOP algorithm [5].

This algorithm was also compared to technique presented in [2] for one of the single output functions, **`9sym.pla`**, and it was found that the number of DSOP forms obtained using the new algorithm (148) is lesser than those obtained in [2] (166).

Table 1: Comparison of results produced by the new algorithm, *espresso* -Ddisjoint and DJ

Name	Size	DSOP	Espresso -Ddisjoint	DJ
xor5.pla	16	16	16	16
co14.pla	14	14	14	14
majority.pla	5	5	10	5
sym10.pla	210	240	367	378
9sym.pla	86	148	209	200
z9sym.pla	86	148	190	186
t481.pla	481	25040	2139	2714

V. Conclusion

These results are encouraging since they show that the number of DSOPs generated from the algorithm described here are better in most of the cases than compared to the earlier algorithms used to compute the DSOP cube list representations of single output Boolean functions.

Future work will focus on refinements and extensions of this technique for multi-output functions. Applications include the use of this method form computing various types of spectral coefficients of Boolean functions and as a preprocessor for various ESOP minimization algorithms.

VI. References

[1] B. J. Falkowski, I. Schafer, M. A. Perkowski, " A fast computer algorithm for the generation of disjoint cubes for completely and incompletely specified Boolean functions," *Proceedings of 33rd Midwest Symposium on Circuits & Systems*, Calgary, Alberta, August 1990, pp. 1119-1122.

[2] B. J. Falkowski, I. Schafer, C. H. Chang, "An effective computer algorithm for the calculation of disjoint cube representation of Boolean functions,", 1993., *Proceedings of the 36th IEEE Midwest Symposium on Circuits and Systems*, Detroit, MI, USA, August 1993, pp. 1308-1311.

[3] Bogdan J. Falkowski, "Calculation of Rademacher-Walsh Spectral coefficients for systems of completely and incompletely specified Boolean functions," *IEEE International Symposium on Circuits and Systems*, Chicago, IL, USA, May 1993, pp. 1698-1701.

[4] B. J. Falkowski and C. H. Chang, "Paired Haar spectra computation through operations on disjoint cubes," *IEE Proceedings on Circuits, Devices and Systems*, June 1999, pp. 117-123.

[5] T. Kozlowski, E. L. Dagless, Jonathan M. Saul, "An enhanced algorithm for the minimization of exclusive-OR-sum-of-products for incompletely specified functions," *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Austin, TX, USA, October 1995, pp. 224-249.

[6] S.L. Hurst, D.M. Miller, J.C. Muzio, **Spectral Techniques in Digital Logic**, London, U.K., Academic Press, 1985.

[7] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, **Logic Minimization Algorithms for VLSI Synthesis**, Boston, MA: Kluwer Academic Publishers, 1984.

[8] M. A. Thornton, R. Drechsler and D. M. Miller, **Spectral Techniques in VLSI CAD**, Kluwer Academic Publishers, Dordrecht, Netherlands, 2001.

[9] T. Kozlowski, Application of exclusive-OR logic in technology independent logic optimization, *Ph.D. Dissertation*, University of Bristol, January 1996.

[10] T. Sasao, Exmin2: A simplification algorithm for exclusive-or sum-of-products expressions for multiple-valued input two-valued-output functions, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(5), pp. 621-632, 1993.

[11] N. Song and M. Perkowski, EXORCISM-MV-2: Minimization and exclusive sum of products expressions for multiple-valued input incompletely specified functions, *Proceedings of the International Symposium on Multi-valued Logic*, pp. 132-137, May 1993.

[12] A. Mishchenko and M. Perkowski, Fast heuristic minimization of exclusive-sums-of-products, Proceedings of the International Workshop on Reed-Muller expansions in circuit design, pp. 242-249, August 2001.