

# Delay-Reduced Combinational Logic Synthesis using Multiplexers

Rekha K. James, Shahana T. K, K. Poulose Jacob  
Cochin University of Science and Technology  
Kochi, Kerala, India  
E-mail: {rekhajames, shahanatk, kpj}@cusat.ac.in

Sreela Sasi  
Gannon University  
Erie, PA, USA  
sasi001@gannon.edu

**Abstract** - *This paper presents an approach to obtain reduced hardware and/or delay for synthesizing logic functions using multiplexers. Replication of single control line multiplexer is used as the only design unit for defining any logic function specified by minterms. An algorithm is proposed that does exhaustive branching to reduce the number of levels and/or modules required to implement any logic function. The algorithm identifies a single variable or a function at the control input of the multiplexer which leads to an implementation with reduced number of levels and/or hardware. Simulation is done upto 9 variable functions using two levels. The approach attains a reduction in delay and/or power over other implementations of functions having larger number of variables. Theoretically, the algorithm can handle completely specified functions of any number of variables.*

**Keywords:** Logic synthesis, universal logic modules, tree network, exhaustive branching, delay reduction.

## 1 Introduction

VLSI implementations using only one type of modular building blocks can decrease system design and manufacturing cost. A multiplexer realization is a natural choice from the viewpoint of cost and speed.

The use of multiplexer as Universal Logic Module (ULM) for realization of logic functions has already been explored by researchers. An algorithm was developed by A. Pal [1] to obtain single multiplexer realization of logic functions with a minimal size multiplexer. The limitation of this approach is that the size of the module changes with changes in function to be realized due to a non-modular implementation. An iterative method for cascade realization using 1-control line multiplexer was presented by R. K. Gorai [2]. The method terminates if the function is not cascade realizable. Further, as the number of variables increases, the number of levels also increases drastically, resulting in increased delay. A programmed algorithm that implements logic functions using tree

structure was presented by A. E. A. Almaini [3]. This algorithm does not explore all the possible branching options of the tree structure, and hence the delay of the circuit synthesized may not be minimal. Also, it does not guarantee the global optimality in all cases. Later A. H. Aguirre [4, 5] proposed a Genetic programming approach to synthesize logic functions using multiplexers. Even though the number of multiplexers used in the delivered circuit had an improvement over standard implementation, the circuit was not minimal or optimal.

In this research, a novel tree-structured exhaustive branching network using 1-control line multiplexer is analyzed for implementing logic functions described by minterms, that reduces delay and/or hardware. A tree network is very suitable for VLSI realization because of the uniform interconnection structure and the repeated use of identical modules. A logic function with  $n$ -variables can be implemented using  $2^n-1$ , 1-control line multiplexers in  $n$ -levels in standard implementation. Any implementation using less than  $2^n-1$  number of modules and/or lesser number of levels can be considered as an improvement in cost and/or speed.

The organization of this paper is as follows: Initially, the problem is described and then the proposed algorithm is demonstrated using examples. Finally, a comparison in terms of delay and number of modules is done for designs using standard implementation and tree network implementation [3] for several functions.

## 2 N-ary Exhaustive Branching Technique

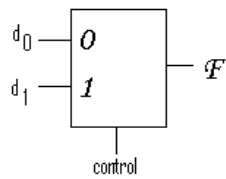
For a given number of input variables  $n$ , there is a well-defined number of functions, which is equal to  $2^{(2^n)}$  [6]. Standard implementation of a tree network requires  $n$ -levels to implement these functions. Almaini proposed a programmed algorithm to reduce the complexity of the network in terms of number of modules and levels. In his

approach 1's, 0's,  $x_i$  (where  $x_i$  is a variable  $x_i$  or its complement,  $1 \leq i \leq n$ ) or functions using any number of variables can be given to any data input of the multiplexer. But the control inputs accept variables only. In this research, the performance is further improved by an exhaustive branching technique using variables or functions at control input. Since functions are also given to control input, the utilization of all branching options are made possible. This decreases the number of levels, and hence the delay is reduced for any logic function implementation using multiplexers.

The first level (output stage) will have a single module, the second level will have a maximum of  $(2^c+c)$  modules, where  $c$  is the number of control inputs ( $c=1$  in this case) and so on. In general, the maximum number of modules in a level can be expressed as  $(2^c+c)^{(L-1)}$  where  $L$  indicates the number of levels. The maximum number of modules in the complete network having  $L$  levels will be

$$\sum_{x=1}^L (2^c + c)^{(x-1)}$$

A network with 1-level can realize functions up to 3 variables, since there are 3 inputs as shown in Figure 1. By connecting  $x_i$  to the control input, the remaining  $x_j$  variables ( $j \neq i$ ) or constants (0 or 1) can be connected to each of the 2 data input lines. So there are 6 possible values for each data input line, resulting in  $6^2$  combinations. Selecting 1 variable as control input from the total of 3 variables, can take  $3C_1$  combinations. Hence a total of  $6^2 \times 3C_1$  combinations are possible by using level 1 out of which 62 are distinct functions. Among these, 24 are 3 variable functions which requires 3 levels in standard implementation and 2 levels in tree implementation.



**Figure 1. Single control line multiplexer module**

Level 2 allows the implementation of functions having maximum of 9 variables, using 3 control lines and 6 data lines. Selecting 3 variables from the total of 9 variables, results in  $9C_3 \times 3!$  combinations. The remaining 6 variables and its complements or constants (0 or 1) at 6 data lines give rise to  $14^6$  functions. Hence  $14^6 \times 9C_3 \times 3!$  combinations are possible at this level. In the tree structure given by Almaini [3], at level 2, maximum number of variables possible is only 7, which results in  $10^4 \times 7C_3 \times 3!$  combinations. The proposed approach increases the number of variables and functions that can be implemented in level 2 itself. As

the levels increases this difference becomes more and more significant and hence reduction in delay can be achieved especially for functions with large number of variables. In general, with  $L$  levels the number of combinations possible in the proposed method is

$$\{ [2(y+1)]^y \} \times \{ z^L C_z^{L-1} \} \times \{ z^{L-1}! \}$$

where  $y = [z^L - z]$  and  $z = 2^c + c$

Maximum number of variables at level  $L$  is

$$n_{\max} = z^L$$

For a given function if there are  $n$  dependent variables, the levels  $L$  required for implementation is given as

$$\lceil \log_{(2^c+c)} n \rceil \leq L \leq (n-1)$$

Whereas in the tree structure,  $L$  can be in the range

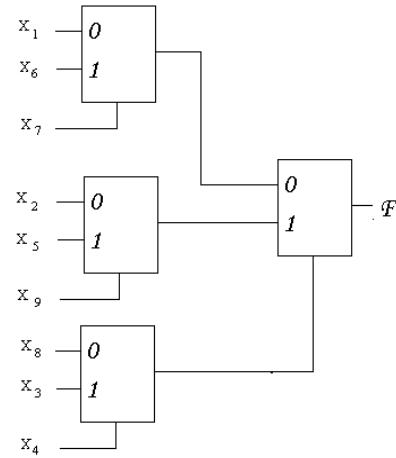
$$\lceil \log_{(2^c)} n \rceil \leq L \leq (n-1)$$

This clearly demonstrates a reduction in delay attained by this exhaustive branching technique. The following example illustrates the delay improvement achieved for a 9 variable function.

Consider the function,

$$F = x_8 x_7 x_3 x_1 + x_7 x_4 x_3 x_1 + x_8 x_7 x_4 x_1 + x_7 x_6 x_4 x_3 + x_8 x_7 x_6 x_4 + x_8 x_7 x_6 x_3 + x_9 x_8 x_4 x_2 + x_9 x_8 x_5 x_4 + x_9 x_4 x_3 x_2 + x_9 x_5 x_4 x_3$$

This function is implemented by a network of 4 modules using the proposed exhaustive branching technique that reduces the number of levels to 2, and is shown in Figure 2.



**Figure 2. Implementation of a 9-variable function using exhaustive branching technique**

## 2.1 Exhaustive Branching Technique

Behavior of a 1-control line multiplexer can be expressed as  $F_s' F_j + F_s F_k$ , where  $F_s, F_j, F_k$  are functions of  $t$  variables ( $1 \leq t \leq n$ ). The number of

variables of  $F_s$ ,  $F_j$  and  $F_k$  varies according to the complexity of the function to be realized. The maximum number of variables in  $F_s$ ,  $F_j$  or  $F_k$  determines the delay of the network. The network terminates when  $F_s$ ,  $F_j$  and  $F_k$  are 1's, 0's or  $x_i^*$  ( $1 \leq i \leq n$ ). If all inputs except one terminates with a variable  $x_i^*$  or a logical constant and only one input continues into the next level, a cascade is generated where a single module is used in each level.

The proposed algorithm aims to identify variables or functions of 2 variables at each control input, that eliminate as many branches as possible, and reduce the number of levels and modules required. This algorithm requires minterms as well as the number of variables of the function in order to implement the exact function. For example the minterms  $\langle 0, 1, 2, 3 \rangle$  result in  $F = 1$  for a two variable function; but for a 3 variable function  $F = x_3$  where  $x_3$  is the most significant variable.

The algorithm for exhaustive branching technique for a function given by minterms is as follows:

Step 1: Get the minterms and number of variables  $n$ , of the given function. Set the level,  $L = 1$  and number of modules  $M = 1$ .

Step 2: List the minterms as minterm table. Check whether number of variables prior to  $L$ ,  $n - (L - 1) \leq 2$ . If so, the tree is completed with any choice of remaining variables, and terminate.

Step 3: Check if there is any variable  $x_i$  for which the number of occurrences, say  $x_c$  in the minterm table is 0 or  $2^{n-L}$ .

For  $x_i = 1$ , if  $x_c = 0$  then data input 1 ( $d_1$ ) = 0  
else if  $x_c = 2^{n-L}$  then data input 1 ( $d_1$ ) = 1.

For  $x_i = 0$ , if  $x_c = 0$  then data input 0 ( $d_0$ ) = 0  
else if  $x_c = 2^{n-L}$  then data input 0 ( $d_0$ ) = 1.

If both data inputs are constants, terminate.

Step 4: Check if there is any variable  $x_i$  for which number of occurrences, say  $x_c$  in the minterm table is 0 or  $2^{n-L-1}$ . If so, check whether there is some variable  $x_j$  ( $j \neq i$ ), for which  $x_j$  remains constant for that  $x_i$ . If so, terminate.

Step 5:  $L = L + 1$ ,  $M = M + 1$ . Get the reduced minterm tables for each variable and find the  $x_i$  for which the following conditions are satisfied.

- (i) One reduced minterm table corresponds to a constant (0 or 1) or  $x_j^*$  ( $j \neq i$ )
- (ii) The other reduced minterm table is a single module implementation by repeating step 4

Step 6: Get reduced minterm tables for each possible ( $x_i \oplus x_j$ ) or  $x_i^*x_j^*$  and check whether the reduced minterm tables corresponds to constants or  $x_k^*$ . If so, terminate.

Step 7:  $M = M + 1$ . Get the reduced minterm tables for each variable, and find the  $x_i$  for which the reduced

minterm tables are single module implementations by repeating the step 4.

Step 8: Get the reduced minterm tables for each possible ( $x_i \oplus x_j$ ) or  $x_i^*x_j^*$  and check whether the reduced minterm tables corresponds to a variable or constant, and a single module implementation by checking the conditions of step 5.

Step 9:  $M = M + 1$ . Get the reduced minterm tables for each possible ( $x_i \oplus x_j$ ) or  $x_i^*x_j^*$  and check whether both reduced minterm tables are single module implementations by repeating the step 4.

Step 10:  $L = L + 1$  and go to step 2.

The exhaustive branching technique is demonstrated using the following examples.

Example 1:

Implementation of the 4-variable function,  $F = \sum (4, 7, 9, 10, 12, 13, 14, 15)$

Step 1: Minterms = (4, 7, 9, 10, 12, 13, 14, 15),  $n = 4$ ,  $L = 1$ ,  $M = 1$ .

Step 2: Form a 4 bit binary minterm table as shown in Table 1.

**Table 1. 4-bit binary minterm table**

$x_4$	$x_3$	$x_2$	$x_1$
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

Here,  $n = 4$  and  $L = 1$ . So,  $n - (L - 1) \leq 2$  is not satisfied.

Step 3: Let  $i = 4$ .

For  $x_4 = 1$ ,  $x_c = 6$  which is neither 0 nor  $2^{n-L} (=8)$ .

For  $x_4 = 0$ ,  $x_c = 2$  which is neither 0 nor  $2^{n-L} (=8)$ .

Similarly no other variable satisfies the conditions.

Step 4: Consider  $x_2$ .

Then,  $x_c = 4$  which is  $2^{n-L-1} (=4)$ .

But none of the remaining variables are constant for  $x_2$ . Checking the same conditions for other variables it is found that none of the variables satisfies the conditions.

Step 5:  $L=2$ ,  $M=2$ . Consider  $x_3$ .

The reduced minterm table for  $x_3 = 0$  is given in Table 2 and for  $x_3 = 1$  in Table 3 respectively.

**Table 2. The reduced minterm table for  $x_3 = 0$**

$x_4$	$x_2$	$x_1$
1	0	1
1	1	0

**Table 3. The reduced minterm table for  $x_3 = 1$**

$x_4$	$x_2$	$x_1$
0	0	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Here, none of the minterm tables corresponds to the condition (i). So try for another variable.

No other variable satisfies the conditions.

Step 6: Therefore, consider  $(x_2 \oplus x_1)$ . The reduced minterm table for  $(x_2 \oplus x_1) = 1$  is given in Table 4 and for  $(x_2 \oplus x_1) = 0$  is given in Table 5 respectively.

**Table 4. The reduced minterm table for  $(x_2 \oplus x_1) = 1$**

$x_4$	$x_3$
1	0
1	0
1	1
1	1

This table reduces to  $x_4$ , for a 2 variable truth table with  $x_4$  and  $x_3$  as variables.

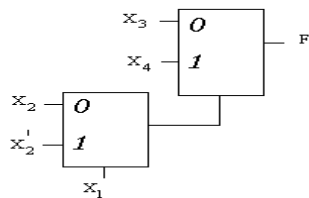
**Table 5. The reduced minterm table for  $(x_2 \oplus x_1) = 0$**

$x_4$	$x_3$
0	1
0	1
1	1
1	1

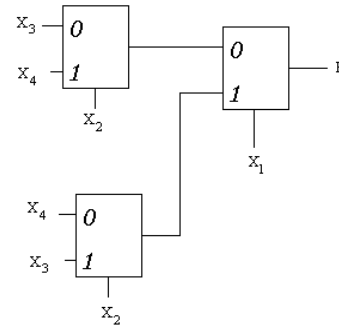
This table reduces to  $x_3$ , for a 2 variable truth table with  $x_4$  and  $x_3$  as variables.

Since all conditions are satisfied, terminate.

The implementation has only 2 modules in 2 levels, as shown in the Figure 3, while in the tree implementation [3], the synthesized network will have minimum of 3 modules in 2 levels as in Figure 4.



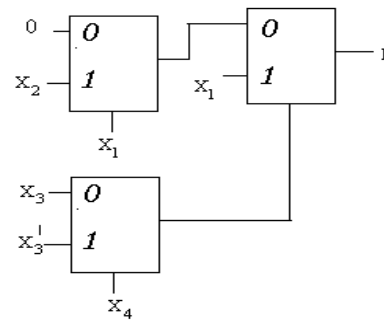
**Figure 3. Exhaustive branched network implementation for  $F = \Sigma(4, 7, 9, 10, 12, 13, 14, 15)$**



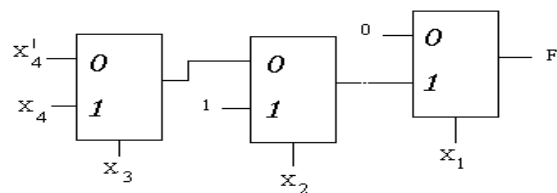
**Figure 4. Tree implementation for  $F = \Sigma(4, 7, 9, 10, 12, 13, 14, 15)$**

It is noted that there is a reduction in number of modules which in turn reduces the power consumption.

Example 2: The implementation of a 4-variable function,  $F = \Sigma(3, 5, 7, 9, 11, 15)$  has 3 modules using only 2 levels in this approach as shown in Figure 5, while the tree implementation [3] requires 3 modules in 3 levels as shown in Figure 6.



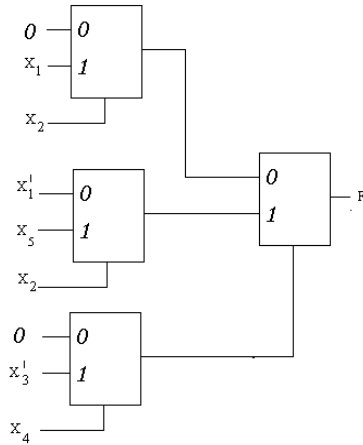
**Figure 5. Exhaustive branched network implementation for  $F = \Sigma(3, 5, 7, 9, 11, 15)$**



**Figure 6. Tree implementation for  $F = \Sigma(3, 5, 7, 9, 11, 15)$**

Example 3:

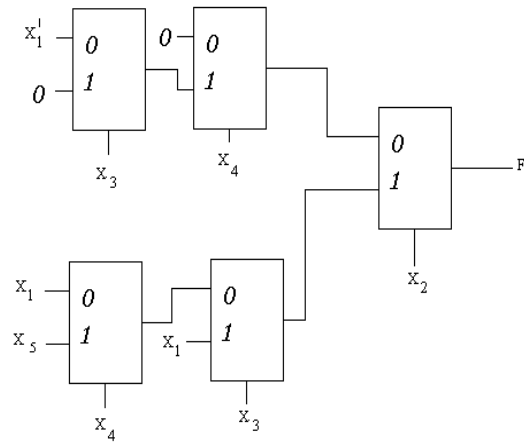
The implementation of the 5-variable function,  $F = \sum(3, 7, 8, 15, 19, 23, 24, 26, 27, 31)$  is given in Figure 7.



**Figure 7. Exhaustive branched network implementation for  $F = \sum(3, 7, 8, 15, 19, 23, 24, 26, 27, 31)$**

The delivered network has only 4 modules in 2 levels, whereas the tree implementation requires more levels using at least 5 modules. One possible implementation is shown in Figure 8.

This example clearly indicates the reduction in delay and hardware for a 5-variable function using the proposed technique.



**Figure 8. Tree implementation for  $F = \sum(3, 7, 8, 15, 19, 23, 24, 26, 27, 31)$**

Simulation is done up to 9-variable functions using two levels. Table 6 shows the reduction in delay and/or hardware for certain functions. The reduction in number of modules required will lead to reduced power consumption.

**Table 6. Comparison in terms of delay and hardware for standard implementation, tree implementation and exhaustive branched network implementation**

Functions	Standard Implementation	Tree Implementation	Exhaustive Branched Implementation
	D / M	D / M	D / M
$F = \sum(1, 2, 4, 7)$	3 / 7	2 / 3	2 / 2
$F = \sum(1, 2, 3, 5, 6, 7, 9, 10, 11, 15)$	4 / 15	3 / 4	2 / 4
$F = \sum(4, 7, 9, 10, 12, 13, 14, 15)$	4 / 15	2 / 3	2 / 2
$F = \sum(3, 5, 7, 9, 11, 15)$	4 / 15	3 / 3	2 / 3
$F = \sum(3, 7, 8, 15, 19, 23, 24, 26, 27, 31)$	5 / 31	3 / 5	2 / 4

D / M – Delay (number of levels) / Number of multiplexers

### 3 Conclusion and future work

An algorithm for the synthesis of delay reduced multiplexer network is presented. By suitable selection of variables or functions as control inputs, the number of modules and/or delay are reduced. The reduction in number of modules results in reduced power consumption of the synthesized network. Theoretically, the algorithm can handle any number of variables for any completely specified logic function. Since the number of variables is also given as input, exact functions are realized. The computation time is not always directly proportional to the number of variables, but increases with the complexity of the function to be realized. Since the topology of the delivered network is that of a tree, VLSI implementation of this network requires very few extra work in routing algorithms to redesign or for circuit layout.

This algorithm can be extended for the synthesis of incompletely specified functions. Optimization can be done by checking for identical modules in different levels. Network complexity can be reduced if the modules considered have normal and complemented outputs. Research may be done to consider different size multiplexers for an alternative implementation.

### 4 References

- [1] A. Pal, "An algorithm for optimal logic design using multiplexers", IEEE Transactions on Computers, Vol. 35, No. 8, August 1986, pp. 755-757.
- [2] R.K. Gorai and A. Pal, "Automated synthesis of combinational circuits by cascade networks of multiplexers", IEE Proceedings-E, Vol. 137, No. 2, March 1990, pp. 164-170.
- [3] A.E.A. Almaini, J.F. Miller and L Xu, "Automated synthesis of digital multiplexer networks", IEE Proceedings-E, Vol. 139, No. 4, July 1992, pp. 329-334.
- [4] A.H. Aguirre, C.A.C. Coello and B.P. Buckles, "A genetic programming approach to logic function synthesis by means of multiplexers", Proceedings of First NASA/DOD workshop on Evolvable Hardware, IEEE Computer Society Press, Los Alanitos, California, July 1999, pp. 46-53.
- [5] A.H. Aguirre and C.A.C. Coello "Using genetic programming and multiplexers for the synthesis of logic circuits", *Engineering Optimization*, Vol. 36, No. 4, August 2004, pp. 491-511.
- [6] V.P. Correia and A. I. Reis, "Classifying n – input Boolean functions", VII Workshop IBERCHIP 2001, Montevideo, IWS 2001, pp. 58.