Using Decision Diagrams to Design ULMs for FPGAs

Zeljko Zilic and Zvonko G. Vranesic

Abstract—Many modern Field Programmable Logic Arrays (FPGAs) use lookup table (LUT) logic blocks which can be programmed to realize any function of a fixed number of inputs. It is possible to employ logic blocks that realize only a subset of all functions, while the rest can be obtained by permuting and negating the inputs. Such blocks, known as Universal Logic Modules (ULMs), have already been considered for application in FPGAs; in this paper, we propose a new class of ULMs which is more useful in the FPGA environment. Methodology for systematic development of such blocks is presented, based on the BDD description of logic functions. We give an explicit construction of a three-input LUT replacement that requires only five programming bits, which is the optimum for such ULMs. A realistic size four-input LUT replacement is obtained which uses 13 programming bits.

Index Terms—FPGAs, ULMs, BDDs, classification of logic functions, synthesis of logic functions.

1 Introduction

The first commercially available Field-Programmable Gate Arrays (FPGAs) in 1985 had an array of three-input logic blocks, where each block could realize any function of three variables using an 8-bit RAM. Such a block is a lookup-table with three inputs (LUT.3). A decade or two before that, there was a significant amount of theoretical research on Universal Logic Modules (ULMs) [7], [17], [21], which are logic blocks capable of realizing all functions of a fixed number of variables assuming that permutations and negations of variables are provided outside these blocks. Some of the established FPGA families from Actel, Xilinx, and Pilkington use blocks derived from such a concept of ULMs. However, more systematic research on the use of ULM circuits as logic blocks in FPGAs appeared only recently [15], [23], [24], [16]. In this paper, we propose a new type of ULMs for use in SRAM-based FPGAs. Practical designs for three- and four-input LUT (LUT.3 and LUT.4) replacements are presented, together with the methodology to systematically derive such blocks.

ULMs are traditionally defined as blocks with m *general purpose inputs* that can realize any function of up to n inputs, $n < m$, under the assumption that permutations and negations of signals are generated cost-free outside the logic block [21]. While the inversions are, in general, not free in FPGAs, as will be shown in Section 5, the permutations of inputs are free for some routing architectures: In Altera's 8k and 10k series FPGAs [2], all possible input signals can be connected to all the input pins of a logic block. Such architectures with *fully connected* inputs are considered in [5] for routing structures in hierarchical FPGAs [1]. The ULM blocks achieve their functionality by *bridging* some inputs and/or *assigning* them to a constant; these are assumed to be costless operations. This concept is illustrated in Fig. 1a. Classical ULM research was based on this definition of ULMs. Lower and upper bounds are known for m as a function of n, and they asymptotically approach each other. To realize all n-input functions, the total number of inputs m needed is on the order of $2^n / log(n)$. Several methods have been proposed for constructing such ULMs [7], [17], [21].

Recent research on ULMs has been focused on investigating the trade-off between the functionality of logic blocks and their usefulness in real applications. The research presented in [15] and [23] attempted to find a subset of functions that a ULM can realize so that it behaves as close as possible to the LUT. These papers deal with blocks that have functionality comparable to LUT.3 [15] and LUT.4 [23], but they are not functionally complete.[1] The block in [15] has four inputs and realizes 10 out of 14 nonequivalent three-input functions, while the block in [23] requires eight inputs to realize almost all four-input functions.

In this paper, we propose a more practical type of ULMs. It is known that adding pins to logic blocks in realistic FPGAs is very costly [18]. Since a total of $m = O(2^n / log(n))$ inputs are needed for realization of an n input function in a standard ULM, an unreasonable amount of routing resources may be needed if such blocks are used.[2] In addition to providing the access to all m input pins, the routing network must provide resources for bridging the input pins. There are $O(m^2)$ bridging connections possible for each block. These are the reasons why, in [15], the total number of inputs is limited to four, as opposed to eight as in [23]. We propose a class of ULM circuits that avoids this problem and limits the number of input pins to n by using separate

• Z. Zilic is with the Department of Electrical and Computer Engineering, McGill University, McConnell Engineering Building, 3480 University Street, Montreal, Quebec, Canada H3A 2A7.
• Z.G. Vranesic is with the Department of Electrical and Computer Engineering, University of Toronto, 10 King's College Road, Toronto, Ontario, Canada M5S 3G4. E-mail: {zeljko, zvonko}@eecg.toronto.edu.

Manuscript received 17 Mar. 1997.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 104677.

1. The block in [15] is named "semi-ULM" to express the fact that it is not functionally complete.
2. It was noticed early [21] that the standard ULMs are not very practical because of the large number of input pins needed.

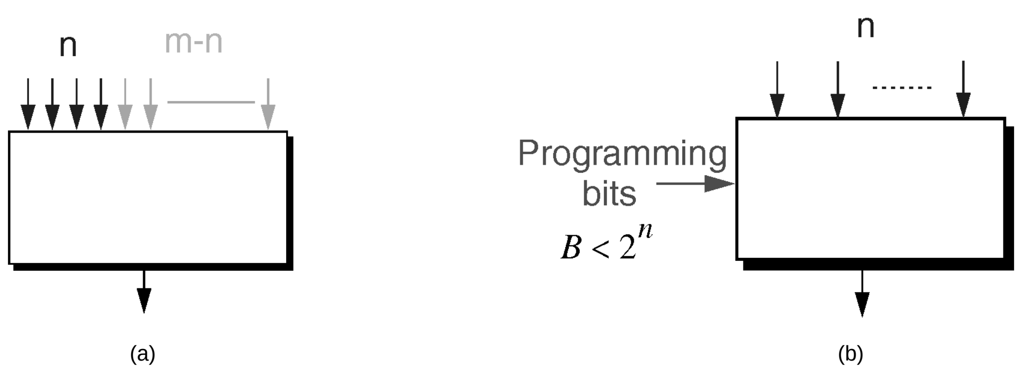


Fig. 1. ULM alternatives. (a) Conventional ULM. (b) Our ULM.

programming bits. Like LUTs, these ULMs are programmed to perform a particular function. As in classical ULMs, the functions obtained by permuting inputs and negating inputs (and, possibly, outputs) are considered to be equivalent. Such ULMs can serve as LUT replacements that require fewer programming bits. Fig. 1 illustrates the difference between these two approaches for a ULM.n that can realize all n-variable functions.

Since the programming bits are loaded, often serially, in SRAM-based FPGAs, there are no additional inputs required, other than the usual function inputs, which would compete for valuable routing resources. When compared with standard LUTs, our blocks need less time and storage area to reconfigure. This is especially important for emerging architectures in which easy reconfigurability of FPGAs is essential [3], [14], [22]. The logic block presented in [22] contains memory storage for four "contexts," which are the programs for the logic block that can be used interchangeably. In this case, any saving in the number of programming bits is multiplied by four.

We, therefore, aim to reduce the *length of the description* of Boolean functions and to develop logic blocks which can use the reduced descriptions as their programs. We show that, for large blocks, it is impossible to obtain a significantly more succinct representation than the one used in a RAM-based LUT. However, for smaller (but practically useful) blocks, savings achieved in the number of programming bits needed can be significant, and we explicitly construct ULMs that reach the theoretical minimum.

The rest of the paper is organized as follows. Section 2 gives an overview of the method used, together with the bounds on the number of bits required in the general case. Sections 3 and 4 provide the explicit designs of ULM.3 and ULM.4 using the proposed methodology. Section 5 gives insight into practical issues related to the use of such ULMs, as well as some extensions of the model.

2 Realization of ULMs

We now describe a procedure for obtaining a class of ULMs with the functionality comparable to LUTs. We exploit the fact that only a subset of all n-variable functions is sufficient to represent them all if inversions and permutations of signals are available. Further, if there are C such functions, they can be encoded by $B = \lceil log_2(C) \rceil < 2^n$ bits. This alone is not attractive if the ULM circuit is too complex or too slow to be of practical use. In this paper, we propose such circuits which are inexpensive relative to the LUTs.

2.1 Equivalence Classes of Switching Functions

The fact that many functions are equivalent under permutation or inversion of inputs and inversion of outputs allows us to group all functions into *equivalence classes*. The equivalence under all three of these operations is commonly called *npn-equivalence* [13]. In FPGAs, we are primarily interested in the restricted notion of *np-equivalence*, which allows permutations and inversions of inputs only. The output inversions and the npn-class will be considered here primarily as a shortcut in developing the main results.

The equivalence classes of switching functions have been investigated in early studies of switching functions [12]. Using enumeration techniques of Polya theory, a closed form expression can be derived for the number of equivalence classes, as a function of n, the number of variables. For our purposes, it is sufficient to derive a lower bound on the number of npn-equivalence classes:

$$C(n) \geq \frac{2^{2^n}}{n! * 2^n * 2}. \tag{1}$$

This bound is obtained as follows: There are at most $n! * 2^n * 2$ different permutations and negations of inputs and outputs, which defines an upper bound on the class size. The number of classes is then larger than the ratio of the number of all possible functions (2^{2^n}) and this bound on the class size. The exact number of equivalence classes is larger than this estimate, especially for small n. For example, for $n = 3$, there are 14 such classes, while, for $n = 4$, the number of equivalence classes is 222.

Our concept of the ULM assumes that a number of programming bits are provided that specify which equivalence class is to be realized by the block. For this model, we can derive an estimate on the number of programming bits needed, $B = \lceil log_2(C) \rceil$. After applying the Stirling approximation [11] for the factorial function and taking a logarithm of (1), we obtain:

$$B(n) \geq 2^n - n\ log(n) - (n+1) + n\ log(e) - 1/2\ log(2\pi n),$$

or

$$B(n) \geq 2^n - n\ log(n) - O(n).$$

TABLE 1
NUMBER OF BITS NEEDED TO ENCODE ALL FUNCTIONS OF n VARIABLES

n	3	4	5
npn classes	14	222	616,126
Bits - npn ULM	4	8	20
Bits - np ULM	5	9	21
Bits - LUT	8	16	32

This bound rapidly approaches the size of the original lookup table 2^n. Hence, for large n, this type of ULMs does not offer much savings in the number of programming bits.

However, for small n, which is of most practical interest, substantial savings can be obtained. For $n = 3$, the total number of bits needed to encode functions realized by a ULM is four (as opposed to eight in LUT.3), while, for $n = 4$ and $n = 5$, the minimal number of programming bits B is eight and 20, respectively. For any number of inputs n, up to $C(n)$ classes of functions have to be provided, for which it is sufficient to have $B(n)$ programming bits. For np-equivalence type of device, one programming bit must be added, which can invert the polarity of the output. Table 1 gives the number of bits required for the ULM.3 through ULM.5 and compares them with the number of bits needed in the corresponding LUT.

Although the saving in the number of programming bits looks encouraging, the implementation of such ULMs may be much larger and slower than that of LUTs. We now derive ULMs whose implementations are comparable to those of LUTs.

2.2 Realization of ULMs Using BDDs

Each equivalence class can be represented by one function, which is called a *class prototype* or a *representative* in the literature. To generate all functions of n variables in a ULM, it is sufficient to have a block that realizes only the representatives.

We devise effective ULMs by constructing a flexible "supercircuit" that can implement all representative functions by using special programmed switches provided in that circuit. Our realization uses the structure of BDDs (Binary Decision Diagrams) [6] to realize a complete set of representative functions. Additional switches, which select the function to be realized, are used to reconfigure the BDD structure. The BDDs are chosen because they are a canonical representation of binary functions which can be used in physical implementation. To realize a function given by a BDD, it is sufficient to replace each node by a multiplexer.

The procedure for designing optimal ULMs consists of:

- enumerating all classes of functions,
- realizing each class representative by a BDD,
- creating a superset structure, called a *Super BDD*, from the union of all BDD representatives,
- providing the flexibility in the Super BDD by adding routing resources,
- minimizing the number of routing paths and switches in the Super BDD,
- minimizing the number of programming bits, and
- optimizing the circuits that use the programming bits to configure the desired function.

TABLE 2
EQUIVALENCE CLASSES FOR FUNCTIONS OF THREE VARIABLES

No.	Class representative	Functions
1	$x_1x_2x_3$	16
2	$x_1x_2x_3 + \bar{x}_1\bar{x}_2\bar{x}_3$	8
3	$x_1x_2 + x_1x_3$	48
4	$x_1x_2 + \bar{x}_1\bar{x}_2x_3$	48
5	$x_1x_2\bar{x}_3 + x_1\bar{x}_2x_3 + \bar{x}_1x_2x_3$	16
6	$x_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + x_1x_2x_3$	2
7	$x_1x_2 + x_1x_3 + x_2x_3$	8
8	$x_1\bar{x}_3 + x_2x_3$	48
9	$x_1x_2x_3 + x_1\bar{x}_2\bar{x}_3$	24
10	$x_1x_2 + x_1x_3 + \bar{x}_1\bar{x}_2\bar{x}_3$	12

To enumerate the function classes for our purposes, it is sufficient to consider only the functions that depend on exactly n variables. The proof that these functions are sufficient is as follows: Assume without loss of generality that a desired function f depends on $n - 1$ variables, $x_1, x_2, \ldots, x_{n-1}$. Then, the logical AND function: x_nf depends on exactly n variables. Since all n-variable functions are represented, it follows that we can realize the function f by assigning the value 1 to x_n. Hence, all other functions depending on the smaller number of variables can be implemented as well.

The step of realizing the class representatives consists of a straightforward implementation of the BDD construction algorithm. The creation of the Super BDD structure is explained in the following two sections, including the simplification of such a structure. In the latter step, we exploit the freedom in selecting any class representative which simplifies the final structure. While we know of no approach better than the search through the class representatives, we notice that using the lexicographically first (based on the function output) class representative, as in [10], a good initial Super BDD is obtained. The final two optimization steps can be achieved by the standard *input encoding* techniques [26]. Based on this approach, we explicitly design three- and four-input logic blocks.

3 REALIZATION OF ULM.3

It is sufficient to enumerate only the npn-equivalent functions, because the BDDs describing both a representative of such class and its complement have the same structure with only the terminal nodes being reversed. For $n = 3$, there are 14 equivalence classes, of which 10 are functions of exactly three variables. Table 2 shows these classes and the number of functions they represent.

For each representative three-variable function, a BDD can have up to five nonterminal nodes. The union of these classes will, therefore, have at most five of these nodes, plus an interconnection structure and programming switches needed to program the ULM.3. All 10 representative BDDs are shown in Fig. 2. The input (control) variables are ordered as: x_1, x_2, x_3; we say that the nodes controlled by variable x_i belong to *level i*. The left outgoing edge, *0-edge*, of

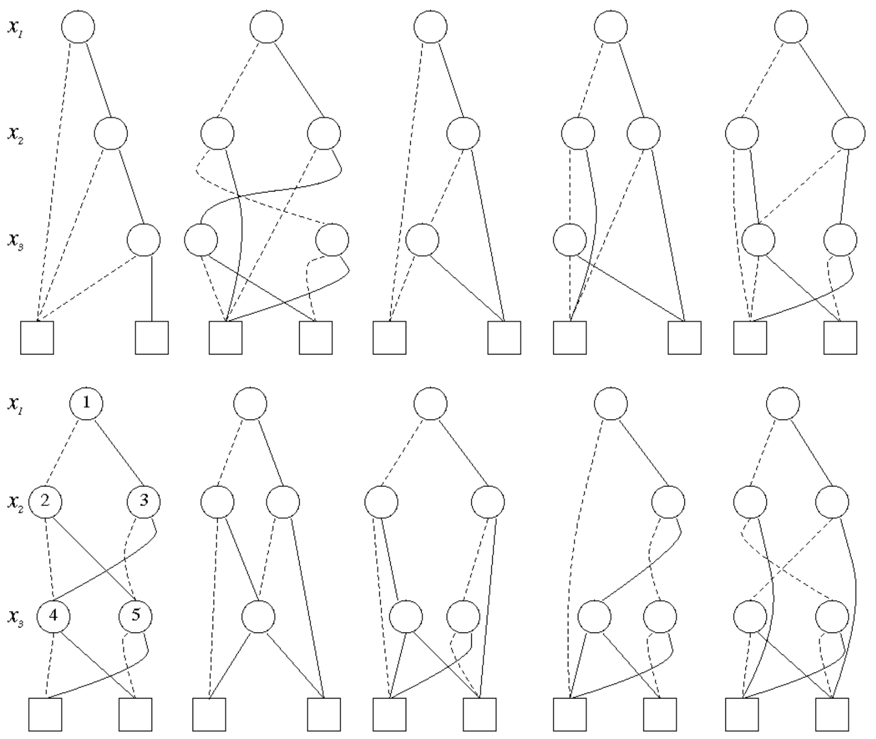


Fig. 2. All representative BDDs for three-variable functions.

each node is taken when the input variable is 0, which is indicated by a dashed line. The two successors are referred to as *0-* and *1-successor*. The values of terminal nodes are not specified because of the possible inversion of outputs under the (considered) npn-equivalence model; it is assumed that, in the canonical case, the left terminal node is zero.

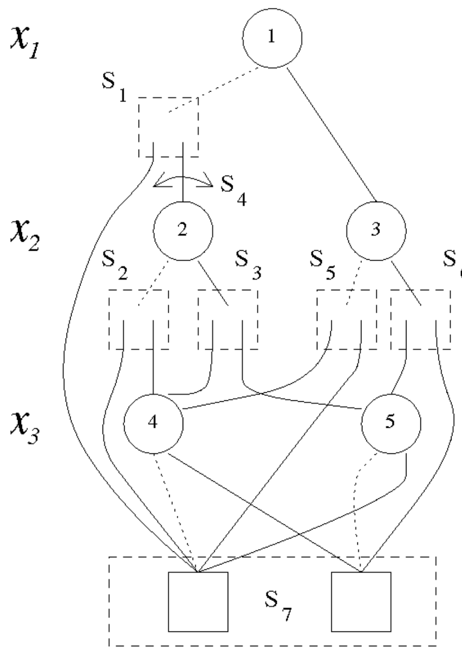


Fig. 3. Super BDD (SBDD.3).

3.1 Super BDD as a ULM

We can combine the BDDs to create the Super BDD (SBDD.3) in Fig. 3 that is capable of realizing all 10 class representative functions of three variables. This union structure has five nonterminal nodes, which we label as in the sixth BDD in Fig. 2.

The SBDD.3 is obtained from the representative BDDs by the following transformations: By enumerating the outgoing edges from each node, a set of possible interconnections is obtained. Sets of outgoing edges are reduced by considering all possible polarities of input variables. At the last stage of optimization, we allow one extension to canonic BDDs, which leads to simpler representation: Polarity of the selection variable at node 2 can be changed independently from the selection variable at node 3. The required polarity change is controlled by the switch S_4 in Fig. 3. Other switches in this BDD are used to define the multiplexed connections to outgoing edges. Switch S_7 changes the polarity of terminal nodes, i.e., it inverts the function.

The design in Fig. 3 requires six switches if npn-equivalence is assumed. The seventh switch, S_7, is needed for the np-equivalence, in which case the terminal nodes can have two possible sets of values. The SBDD.3 can implement the functions of two or one variables by assigning some inputs to a constant.

The SBDD.3 can be used as a ULM. Using one bit per switch, the number of programming bits is seven, which gives a saving of one bit compared to LUT.3.

3.2 Encoding of Programming Bits

It is possible to shorten the function descriptions for the proposed ULM by encoding all possible configurations of switches more compactly. To optimize the encoding of

TABLE 3
PROGRAMMING BITS
FOR THREE-VARIABLE REPRESENTATIVE FUNCTIONS

Function Number	Programming Switch 1	2	3	4	5	6	Inverted Variable
1	0	x	x	x	1	0	x_3
2	1	0	0	1	1	0	x_3
3	0	x	x	x	0	1	
4	1	0	0	1	1	1	
5	1	0	0	0	0	0	
6	1	1	1	1	0	0	x_3
7	1	0	0	0	0	1	
8	1	0	1	0	0	1	x_2 and x_3
9	0	x	x	x	0	0	one of x_2, x_3
10	1	0	1	1	0	1	

programming bit patterns, we enumerate all possible switch assignments for each representative function. The programming bit settings for each function are listed in Table 3. The left path of a programming switch is selected if the corresponding bit is equal to 0. An exception is switch S_4, which changes the polarity of node 2 in the BDD. If this bit is one, then the 0-successor of node 2 is selected when the input variable x_2 is 1. The last column in Table 3 indicates if any variable is complemented at the input to the circuit. For function 9, either x_2 or x_3 should be complemented, but not both.

The desired optimization can be achieved by using more compact *input encoding*. Since there are 10 programming combinations, four bits are sufficient to encode all of them uniquely. However, the objective is to use the simplest logic circuits for this compact encoding. To achieve a simple realization, we use the functional composition and reordering of some of the BDDs in Fig. 2, rather than the standard input encoding approaches [26].

Switches 1 through 4 in Table 3 can be encoded separately, because the functions can be decomposed. Since there are six combinations of these switches in Table 3, three encoding bits are necessary for these four switches, and five bits are needed for the whole circuit. However, the optimal ULM.3 should use only four bits in total, as shown in Section 2.1. To obtain the optimal length encoding, we can have at most four programming combinations for these four bits. To achieve a further reduction, we consider all input permutations for the representative functions. Only five functions can be replaced by reordering the variables, because the other five functions are symmetrical (indicated by the bold typeface in Table 3). Table 4 lists the programming combinations for the permuted representatives that were used to optimize the SBDD.3. The decoder circuit can be simplified if the two functions in Table 4 replace the corresponding functions in Table 3. Function 10 should be replaced with the permutation which has all programming bits set to one. This permutation has x_1 and x_2 interchanged, and it implements the function $x_1x_2 + x_2x_3 + \overline{x}_1\overline{x}_2\overline{x}_3$. With this function the inversion of the BDD, node 2 may or may not be applied, because the variable x_3 can be independently inverted in the BDD. Second, function 8 is replaced by

TABLE 4
PROGRAMMING BITS
FOR PERMUTED REPRESENTATIVE FUNCTIONS

Function Number	Programming Switch 1	2	3	4	5	6	Inverted Variable
8	1	0	0	1	0	1	x_2
10	1	1	1	x	1	1	

TABLE 5
ENCODING FOR THE FIRST FOUR SWITCHES

B_0	B_1	S_1	S_2	S_3	S_4
0	0	0	x	x	x
0	1	1	0	0	1
1	0	1	0	0	0
1	1	1	1	1	1

TABLE 6
OPTIMIZED PROGRAMMING BITS

No.	B_0	B_1	S_5	S_6	Inverted
1	0	0	1	0	x_3
2	0	1	1	0	x_3
3	0	0	0	1	
4	0	1	1	1	
5	1	0	0	0	
6	1	1	0	0	x_3
7	1	0	0	1	
8	0	1	0	1	x_2
9	0	0	0	0	
10	1	1	1	1	

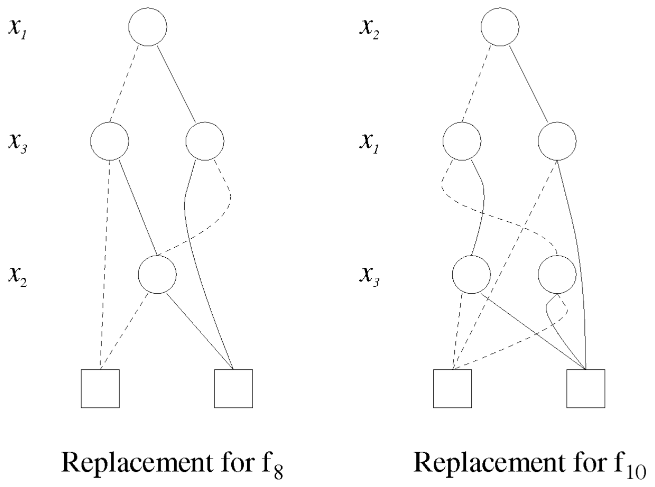


Fig. 4. Two replacement BDDs.

an implementation that has variables x_2 and x_3 interchanged, which leads to the programming vector 100101. The two replacement BDDs are shown in Fig. 4. With these replacements, there is a total of four different combinations of the programmable switches S_1 through S_4. These combinations are given in Table 5. An efficient encoding that satisfies the constraints in the table is given by:

$$S_1 = B_0 + B_1$$
$$S_2 = S_3 = B_0 \cdot B_1$$
$$S_4 = B_1.$$

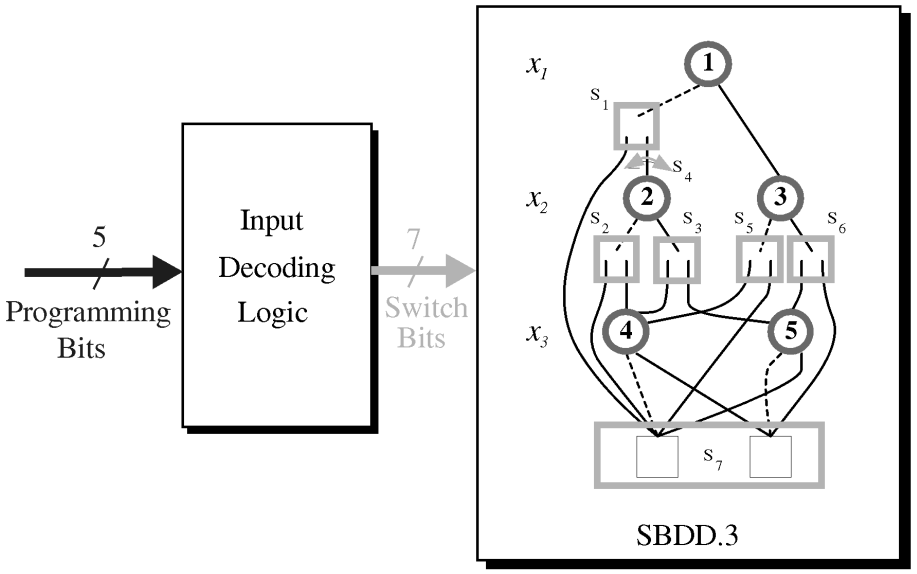


Fig. 5. ULM.3 implemented as SBDD.3 and a decoder.

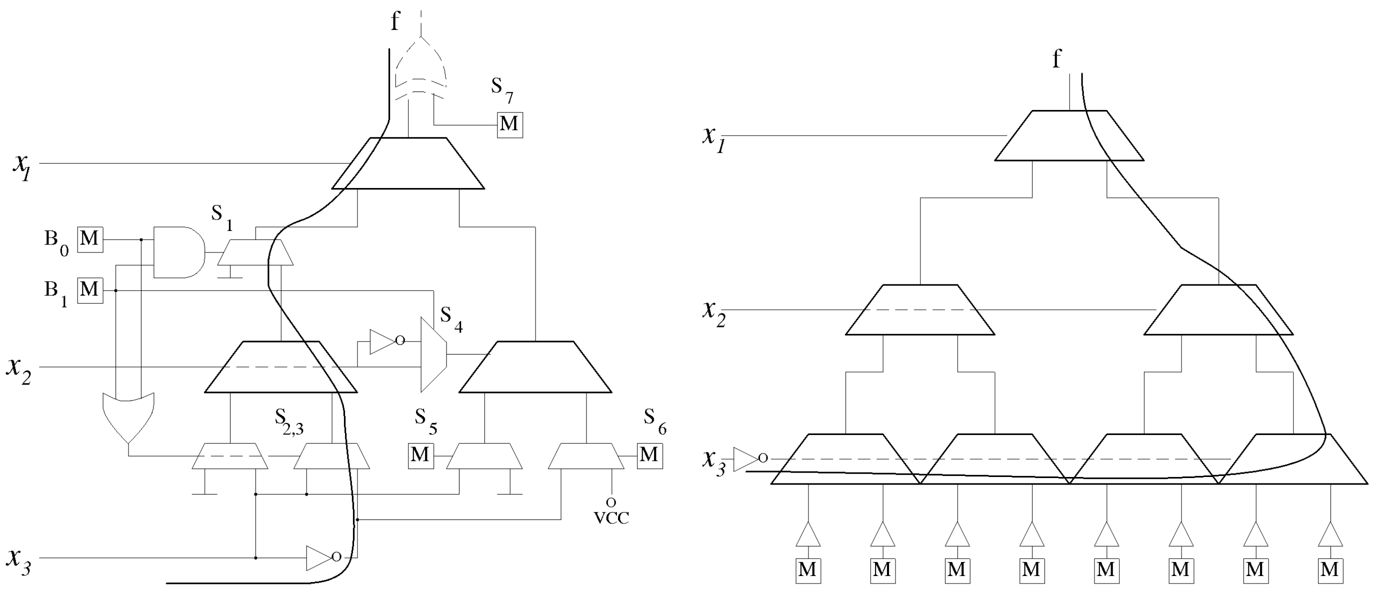


Fig. 6. Implementation of ULM.3 and LUT.3. (a) ULM.3 and its critical path. (b) LUT.3 and its critical path.

This allows a simple implementation of the decoder using only two two-input gates. (Note also that a simple polarity change in switches and/or programming bits B_0 and B_1 allows us to use simpler NAND and NOR circuits.) The programming bits needed after the minimization is performed are shown in Table 6.

3.3 Implementation Issues

The proposed ULM.3 can be implemented in a straightforward way. In addition to the SBDD.3, the decoder and the programming bit memory are needed. The decoder consists of two two-input gates, while the programming bits can be kept in a standard SRAM memory. Fig. 5 shows the overall structure of the ULM.3.

The implementation of ULM.3 can follow the layout of the SBDD.3, which has physically a fairly rectangular shape. Fig. 6 compares its layout with the LUT.3 logic block. Switches in the ULM.3 correspond to those in SBDD.3. We targeted the pass-gate CMOS implementation for most of the logic, as in many LUT implementations [8].

The ULM.3 implementation has advantages with respect to the area. First, only three multiplexers (outlined in bold) in ULM.3 must have the complete functionality; all others perform simpler logic functions, which allows us to decrease the area required without any impact on the speed. Moreover, our ULM avoids a constant overhead of one buffer that each SRAM memory must have when used in LUT configurations [8]. Since there is a possibility of bidirectional current flow when the input to LUT changes,

TABLE 7
COMPARISON BETWEEN LUT.3 AND ULM.3

	LUT.3	ULM.3	Note
Memory (bits)	8	5	4 for dual output
Datapath Mux	7	3	1 invertible
Program Mux	0	5	can be small
Decoder	0	1	two 2-input gates
Program Inv.	0	2	S_4, S_7
Transistors	78	70	5 transistor RAM
Delay	1.38 ns	1.31 ns	small width transistors

the contents of SRAMs can be erased and an inverter must be added to isolate the SRAM cells. In ULM.3, the memory controls only the gates of transistors and there is no current flow toward SRAMs. Finally, the tree-like structure, which, in standard LUT architectures can cause an area overhead in the physical layout, is not present in this block.

The ULM.3 implementation also has some advantages when the delay is considered. Paths of input signals can be kept shorter, which limits the propagation delay through the block. Fig. 6 shows the critical paths for the two circuits, from input x_3 to output f.

Both LUT.3 and ULM.3 in Fig. 6 have been simulated in Spice using 0.8 μm BNR BATMOS technology [4]. With small topology transistors, both blocks (output buffers were not considered) work equally fast. Even though our ULM.3 block uses one more level of logic than LUT.3, the time critical signal, control variable x_3, does not control any pass transistors; instead, this signal propagates through the channels of MOS transistors which are set (opened or closed) long before the signal reaches the transistor. The polarity inversion, which is shown in Fig. 6 as an XOR gate at the output of the circuit, is, in actual implementation, obtained by inverting the terminal nodes; such an inversion does not increase the critical path of the signal. The comparison is summarized in Table 7.

One other variation is possible; the block can have the function output available in both its true and complemented form. It would then be left for the routing resources to select the proper polarity (or both). In this case, only four programming bits would be needed for the block, but adding one more output pin is expensive.

4 LARGER BLOCKS—ULM.4

The same procedure can be used to design ULMs for larger logic blocks, but the complexity of the process increases rapidly. We illustrate the procedure of (computer aided) search for an effective logic block, using the example of ULM.4. It is theoretically possible to derive a four-input block that uses a minimal number of programming bits (eight for npn-equivalent and nine for np-equivalent ULMs). However, since the minimal-length encoding may be too expensive to implement, we investigated the trade-offs between the encoding length and the circuit complexity by using nonoptimal encodings as well.

There are 222 npn-equivalence classes of four-variable functions; 208 of them depend on exactly four variables. These classes are enumerated in [10]. We used this enumeration as an input to a program that generates BDDs for all representative functions. We made these BDDs mutually comparable by using a unique node labeling. The next step was to analyze the connectivity pattern and find the outgoing edges from each node in the BDD. Then, a possible Super BDD structure was constructed, which had a number of programmable switches. The remaining steps consisted of minimizing the number of switches, the number of programming bits, and the logic needed to perform the encoding function for the switches.

4.1 Unifying Representative BDDS

The first step in the procedure for developing the ULM is easy to automate. All class representatives in [10] are sorted according to the output they produce. There are 208 such functions; we will refer to them as f_1 to f_{208}.

The maximum number of nodes in individual BDDs dictates how many nodes there should be in the SBDD.4. Theoretically, the largest BDD representing a four-variable function should have at most nine nodes, excluding the terminal nodes. Starting from the root (level 1) node, the edges can branch as in the full decision tree, except for the fact that there can be only two level-4 nodes. There are 15 BDDs that have this maximal number of nodes.

Unique labeling of nodes is necessary for analyzing the BDDs in a unified way. It is easy to distinguish between two level-2 nodes: they are either 0- or 1-successors of the root node. Also, the two terminal and two level-4 nodes are unique. To label the remaining, level-3 nodes, we use the following scheme. We assign numbers 1 and 2 to terminal 0 and 1 nodes, respectively. Then, a node v is labeled by combining the labels of its 0- and 1-successors using the expression

$$label(v) = 2 * label(v.0) + label(v.1). \quad (2)$$

Fig. 7 shows an example of this BDD labeling. The terminal nodes are labeled as 1 and 2, respectively, while all other labels are obtained using the function *label*. Note that nodes 4 and 5 correspond to the functions x and $\overline{x}$, respectively.

This labeling is almost unique. There are $\binom{4}{2} = 12$ possible combinations of two different successors of level-3 nodes and only two pairs of successor nodes with the same

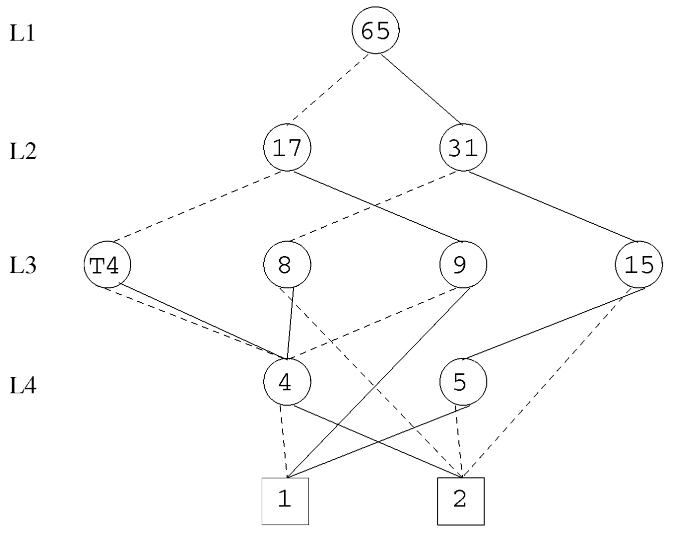


Fig. 7. Example BDD labels.

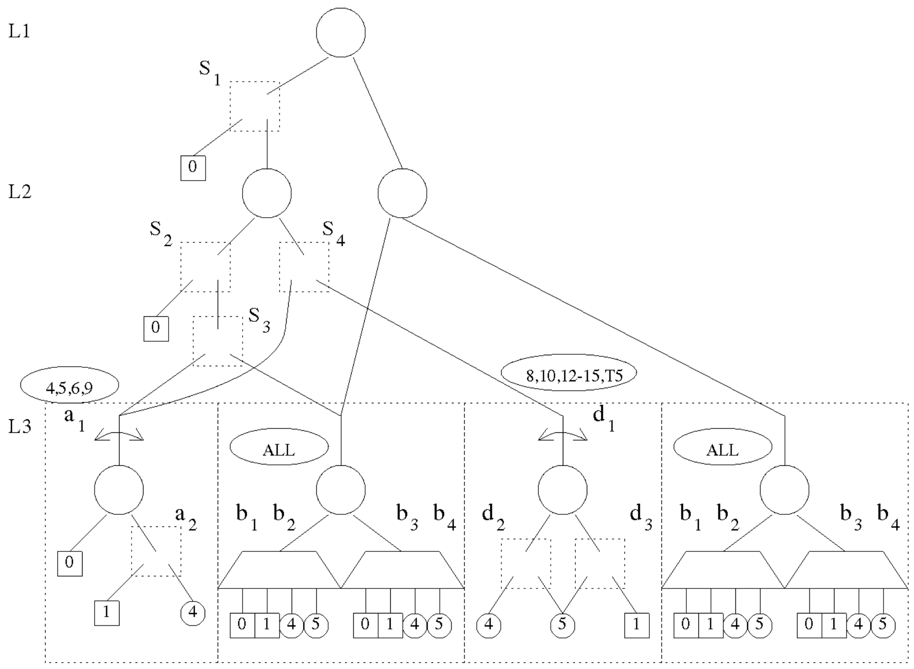


Fig. 8. Outline of SBDD.4.

label. Number 9 can be obtained as 9 = 2 ∗ 2 + 5 = 2 ∗ 4 + 1, hence the number 9 could label two different nodes, the one with successors (2, 5) and another with successors (4, 1). To make the labeling unique, we assign the label 15 to the second of these nodes while keeping the label 9 for the first node, as in Fig. 7.

To optimize the interconnect, we allow that some level-3 nodes can have both outgoing edges pointing to the same node. (Note that this node would not exist in standard BDDs.) When the successor of such node is at level 4 (node 4 or 5), we label the level-3 node as T4 or T5. An example of node T4 is shown in Fig. 7. Note that, when computing a label for a node at level 2 (e.g., node 17 in the figure), the value of T4 is 4.

The interconnect patterns can be analyzed with the above unique labeling. For each node v, the set of successors $S(v)$ is recorded. Since we want to minimize the total interconnect, we first examine if the successor sets can be minimized. We found that, since there are many functions, only a few edges can be eliminated by permuting the variables in some of the functions.

Analysis of the structure of the given representative functions shows much regularity. For example, there is no edge 0 (dashed edge) between the root and the level-3 nodes. The goal is to exploit the regularity in the prototype functions to simplify the structure of the Super BDD.

4.2 Generating Super BDD (SBDD.4)

The number of nodes in the SBDD.4 was determined in the previous step, and it remains to determine the interconnections. The first step in this process consists of assigning the node labels to the physical nodes in the SBDD.4. There are 12 possible labels for the level-3 nodes, but only four such nodes are present in the SBDD.4. Therefore, these nodes must be capable of realizing several functions. The functions should be assigned to these nodes to minimize the total number of switches.

Fig. 8 shows an outline of one possible SBDD.4. The functions to be realized by level-3 nodes are enumerated inside the oval and the circuits that implement these functions are shown in more detail in the associated dotted boxes. The two level-4 nodes, 4 and 5 (functions x and $\overline{x}$) are used as primitives and we did not expand them further. The two terminal nodes, 0 and 1, are indicated inside the boxes.

The total number of switches to control the SBDD.4 is 17, one more than the number of programming bits for LUT.4. One additional bit (a_1 in Fig. 8) was eliminated in the final optimization. Several iterations were made in permuting the variables, to minimize the number of switches. By exchanging variables x_1 and x_2 in functions f_{185} and f_{207}, variables x_1 and x_3 in f_{205}, f_{206}, and ordering the first three variables as x_2, x_3, x_1 in f_{201}, we eliminated two switches, a_1 and d_2, from the SBDD.4. Elimination of d_2 was possible because all four functions needed in L3.4 node can be realized if the left edge always points to node 4. These two optimization steps decreased the number of switches to 15.

The encoding of the switches was analyzed and grouped with respect to the configuration of the first four switches. All functions are enumerated in Table 8. All possible functions assigned to the four L3 nodes are given in columns. While nodes 2 and 3 (called L3.2 and L3.3) have to realize all possible functions, the other two L3 nodes have to realize just a few, which leads to a saving in the number of switches. Thus, the SBDD.4 can be simplified as shown in Fig. 9. This implementation requires 15 programming switches and, hence, 15 programming bits.

TABLE 8
ENCODING GROUPS

G	$S_1S_2S_3S_4$	L3.1	L3.2	L3.3	L3.4	#
G_1	0xxx	x	1, 4, 6, 10, 13	2, 6, 8, 10-15	x	16
G_2	10x0	4, 6	T5, T4, 2, 4-8, 10-15	T5, T4, 2, 1, 4-15	x	93
G_3	10x1	x	2, 10, 12-15	T5, T4, 2, 1, 4-15	10, 13	59
G_4	1101	4, 6	T5, 8, 10-15	T5, 4, 5, 6, 8-15	8, 10, 13, 14	39
G_5	1111	x	14	13	14	1

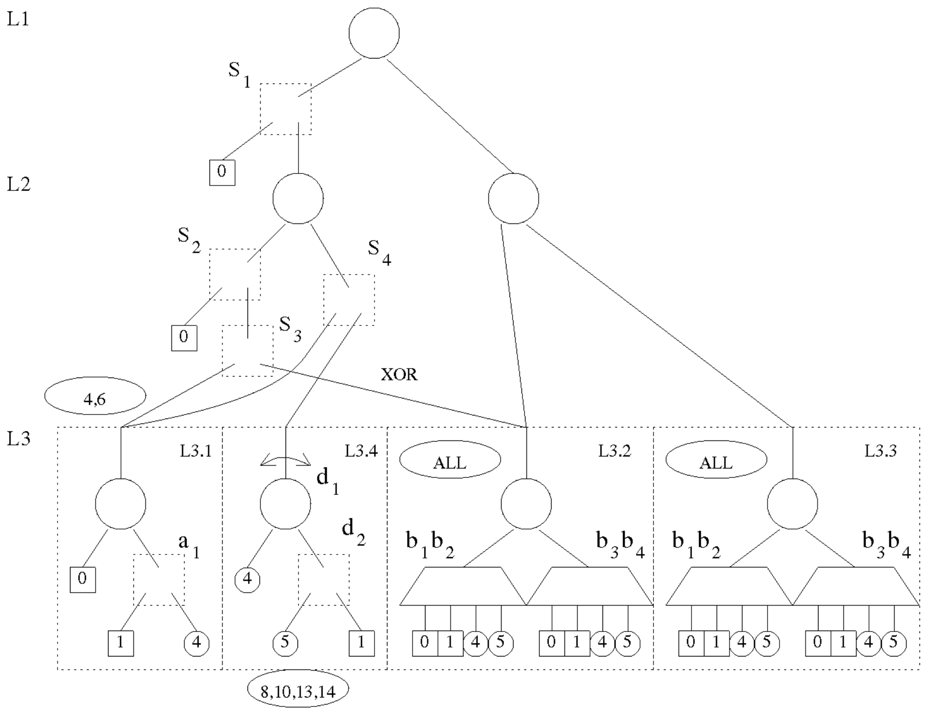


Fig. 9. Optimized SBDD.4.

4.3 Input Encoding for Programming Bits

To reduce the number of programming bits, we can encode the possible switch settings. To achieve this, we used the input encoding algorithm in NOVA, which is included in the SIS [20] package. Since the minimal-length encoding is expected to be expensive, we considered several other encoding strategies.

The four switches S_1 through S_4 can be encoded using three bits for five possible configurations, corresponding to groups G_1 to G_5, as shown in Table 9. This leads to a simple decoder:

$$S_1 = B_3$$
$$S_2 = B_1$$
$$S_3 = B_2 \cdot B_3$$
$$S_4 = B_2 + B_1.$$

Fourteen programming bits are needed in this arrangement. Note that group G_5 in Table 8 has exactly one function in it (the four-input XOR), which costs a programming switch, S_3. The edge emanating from this switch is marked as "XOR" in Fig. 9.

Further reduction in the number of programming bits can be achieved through more careful encoding of functions in each of the existing five groups. Notice that this encoding does not affect the speed of the circuit. All additional circuits are placed between the memory cells and switches, and they are not in the path of the signal.

We were able to remove two more bits in the encoding of the function with less than 10 gates required for all the decoding circuitry in the ULM.4. The encoding of functions is based on sharing the bits among the sub-blocks and encoding the XOR function as part of group G_4. We omit the details of this encoding. The threshold of 10 gates was selected for the decoding circuitry because, with this overhead, our ULM.4 is still smaller than LUT.4. Thus, we can realize a block that uses 12 programming bits with no extra

TABLE 9
ENCODING FOR FIRST FOUR SWITCHES

G	B_1	B_2	B_3	S_1	S_2	S_3	S_4
G_1	0	0	0	0	x	x	x
G_2	0	0	1	1	0	x	0
G_3	0	1	1	1	0	x	1
G_4	1	0	1	1	1	0	1
G_5	1	1	1	1	1	1	1

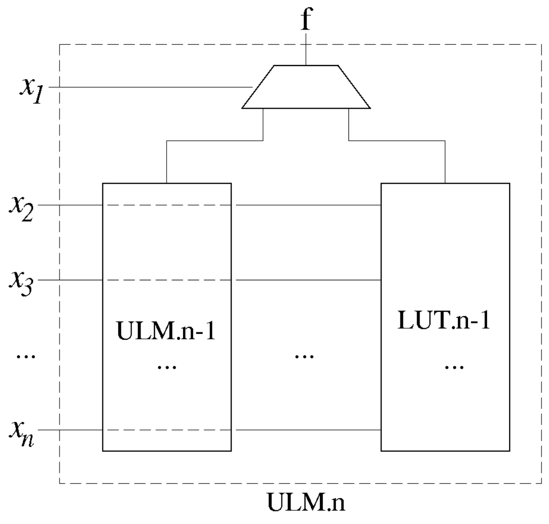


Fig. 10. Construction of larger blocks.

expense in the hardware. To go further, to the theoretical limit of eight bits, one must rely on the general-purpose input encoding programs, combined with pre-encoding. We tried several encoding alternatives, but the circuits that we obtained in this way were too expensive to be used in realistic blocks. It is an open question if there exists a solution that uses less than 12 bits with a reasonable amount of decoding circuitry. Note that all encodings in this section are given for an npn-equivalent/dual-output class of ULMs, and that one more bit should be added for np-equivalent circuits.

4.4 Larger Blocks

The optimization steps required to produce these blocks become costly as the size of the block increases. A simple alternative is to construct ULMs from a smaller size ULM and LUT, as shown if Fig. 10. This would produce a realistic-size ULM.5 which requires 29 programming bits.

5 Other Issues

5.1 Technology Mapping Using ULMs

The ULM blocks presented provide the same functionality as lookup tables, but with a restriction on the ordering and polarity of variables. For recognizing the order and polarity of inputs and outputs of each block, we have implemented an algorithm based on Generalized Reed-Muller form matching, as in [25].

Assuming that the interconnection resources in an FPGA allow arbitrary permutations of inputs to logic blocks, the only remaining constraint is the polarity of input variables to each block. There can be a polarity disagreement when both polarities of a signal are needed. Then, some of the ULM-based blocks must be replicated. The study in [15] found, using the MCNC benchmarks, that their ULM needs both polarities for only 6.6 percent of the nodes. These results do not automatically extend to our block of the same granularity (ULM.3) because that study was done using a block with incomplete functionality and some of the class representatives used differ in the polarity. However, since nodes tend to have smaller fanout when the granularity of the logic block increases, the results for our larger blocks (ULM.4) cannot be worse.

5.2 Functionally Incomplete Blocks

The motivation of work in [15] and [23] was to investigate the construction of functionally incomplete blocks. Their blocks implemented 10 of 14 and 201 of 208 representative functions, respectively. It is interesting to note that using standard approaches for designing a functionally complete ULM.3, the best solution requires five input pins [21]. The only reason why four pins were sufficient in [15] is that the block was incomplete. The price of such complete LUT.3 replacement is obviously too high. Since our ULM.3 is complete and requires a minimum number of programming bits with no area or delay overhead compared to LUT.3, there is no need to consider incomplete blocks for three-variable functions.

The SBDD.4 construction given here is useful in considering incomplete blocks as well. It is obvious from our ULM.4 that eliminating the 4-input XOR function (f_{208}) would remove one switch and one programming bit in the block. Another bit can be saved by considering the node L3.4 in Fig. 9, for which there are four possible functions. Analysis shows that one switch can be eliminated by excluding functions f_{185} and f_{202} to f_{206}. Hence, the logic block based on our ULM.4 that implements all but these seven functions requires 12 programming bits, with the decoding circuits consisting of only two two-input gates. For comparison, the block in [23] realizes the same number of functions, but eight input pins are required.

One more bit can be removed by excluding two more functions, f_{201} and f_{207}, but at a price of one more two-input decoding gate. This incomplete ULM circuit would require 11 bits for a dual-output block and 12 bits for a single-output np-equivalent block and it would realize 199 of 208 class representatives.

These considerations can lead to useful larger blocks as well. By excluding larger sets of functions, compact ULM circuits can be devised. A similar study appeared in [24], which investigated universal logic modules for a class of series-parallel functions which are important with respect to synthesizability. The study presented in [9] evaluated functional capability of some commercial architectures.

5.3 Using Other Graph-Based Representations—FDD Case

The methodology for constructing ULMs presented here can be used in conjunction with several other Boolean function representations. We consider Functional Decision Diagrams (FDDs), which are the BDD representations of the Reed-Muller form [19]. Compared to BDDs whose nodes can be realized by multiplexers, each node of an FDD can be realized by a Boolean function of the type

$$a \oplus bx.$$

Enumeration of all FDD class representative implementations of $n = 3$ and $n = 4$ reveals that the FDD implementations are simpler overall. Fig. 11 illustrates this fact for $n = 3$. While the output negation of a function results in a BDD of the same shape (but reversed terminal nodes), the same operation produces two different FDDs, of which the simpler one can be taken as a class representative for a given class.

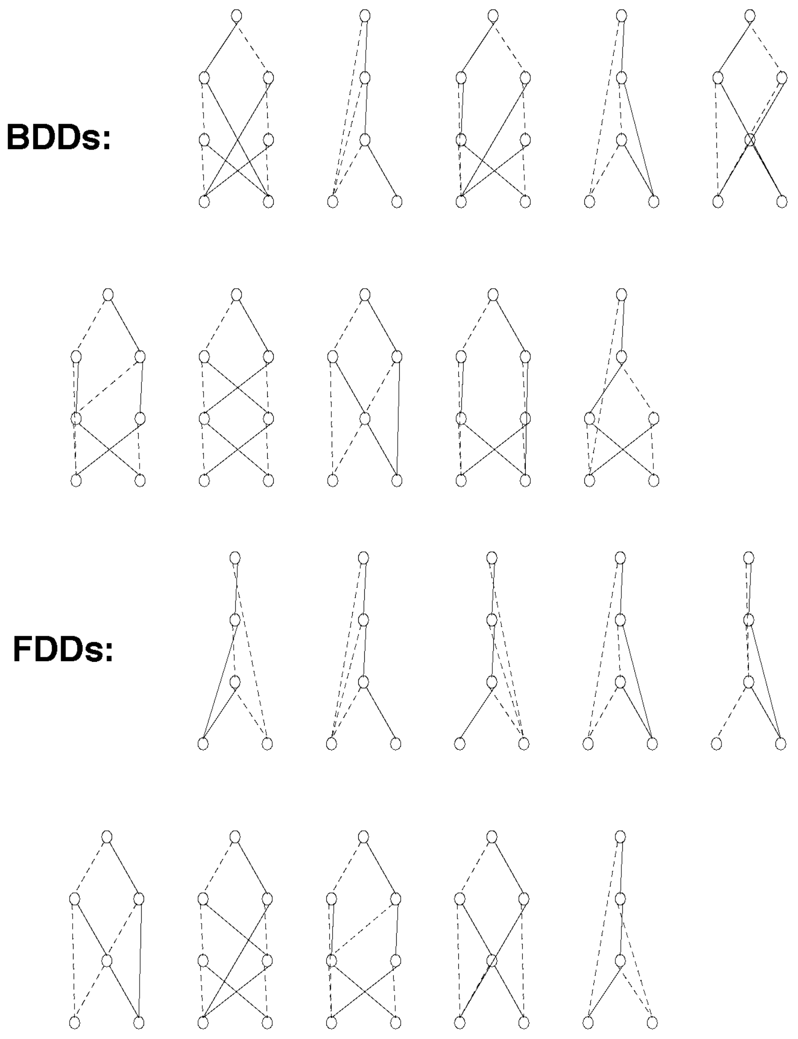


Fig. 11. Comparison of BDDs and FDDs for n = 3.

6 Concluding Remarks

We presented a class of FPGA logic blocks based on the concept of ULMs, which are functionally complete if the permutations and negations of inputs are provided outside the block. As in SRAM-based FPGAs, we use separate programming bits, which is of advantage in practical FPGAs. Previously considered ULMs require costly additional inputs to the logic blocks.

We also presented a methodology for designing such blocks, using decision diagrams and showed detailed designs of replacements for three- and four-input lookup tables. In the case of ULM.3, five programming bits are needed for a block slightly smaller than LUT.3. For ULM.4, several alternatives with different trade-offs between the number of programming bits and the complexity of the circuit were considered. A circuit that requires 13 bits was devised such that it is smaller than LUT.4. An especially important problem in the construction of such ULMs is the input encoding problem, and we used the domain-specific methods which outperform the standard approaches. Furthermore, while the known ULM circuits considered for application in FPGAs [15], [23] are functionally incomplete, our construction offers the complete functionality at a reasonable cost.

The proposed blocks are particularly interesting for FPGAs that will cater to the emerging area of reconfigurable computing.

Acknowledgment

An early version of this paper appeared in the Proceedings of the ACM FPGA ’96 Conference.

References

[1] A.A. Aggarwal and D.M. Lewis, “Routing Architectures for Hierarchical Field Programmable Gate Arrays,” *Proc. Int’l Conf. Computer Design*, pp. 475-478, 1994.
[2] Altera Corp., *Data Book*. San Jose, Calif., 1995.
[3] Atmel Corp., *Configurable Logic Design and Application Book*. San Jose, Calif., 1995.
[4] Bell Northern Research, *Design Rules for CMC 0.8-micron BiCMOS, a Version of BATMOS*. Ottawa, Canada, 1993.
[5] V. Betz and J. Rose, “Cluster-Based Logic Blocks for FPGAs: Area-Efficiency vs. Input Sharing and Size,” *Proc. CICC ‘97*, Santa Clara, Calif., pp. 551-554, Apr. 1997.
[6] R.E. Bryant, “Graph-Based Methods for Boolean Function Manipulation,” *IEEE Trans. Computers*, vol. 37, no. 8, pp. 677-691, Aug. 1986.
[7] X. Chen and X. Wu, “Derivation of Universal Logic Modules, for *n* ≥ 3, by Algebraic Means,” *IEE Proc.*, Pt. E, vol. 128, no. 5, pp. 205-211, Sept. 1981.

[8] P. Chow, S.O. Seo, J. Rose, K. Chung, I. Rahardja, and G. Paez, “Architecture and Circuit-Level Design of an SRAM-Based Field-Programmable Gate Array,” *IEEE Trans. VLSI*, to appear.

[9] J. Cong and Y.-H. Hwang, “Boolean Matching for Complex PLBs in LUT-based FPGAs with Application to Architecture Evaluation,” *Proc. Int’l Symp. FPGAs, FPGA ’98*, pp. 27-34., Feb. 1998.

[10] J.N. Culliney, M.H. Young, T. Nakagava, and S. Muroga, “Results of the Synthesis of Optimal Networks of AND and OR Gates for Four-Variable Switching Functions,” *IEEE Trans. Computers*, vol. 27, no. 1, pp. 76-85, Jan. 1979.

[11] R.L. Graham, D.E. Knuth, and O. Patashnik, *Concrete Mathematics*. Addison-Wesley, 1994.

[12] M. Harrison, “Counting Theorems and their Applications to Classification of Switching Functions,” *Recent Developments in Switching Theory*, A. Mukhopadhyay, ed., pp. 86-121. Academic Press, 1971.

[13] S.L. Hurst, D.M. Miller, and J.C. Muzio, *Spectral Techniques in Digital Logic*. London: Academic Press, 1985.

[14] D. Jones and D. Lewis, “A Time Multiplexed FPGA Architecture for Logic Emulation,” *Proc. Third Int’l Symp. FPGAs, FPGA ’95* , pp. 121-126, Monterey Bay, Calif., Feb. 1995.

[15] C.C. Lin, M. Marek–Sadowska, and D. Gatlin, “Universal Logic Gate for FPGA Design,” *Proc. ICCAD ’94*, pp. 164-168, San Jose, Calif., Oct. 1994.

[16] C.C. Lin, M. Marek–Sadowska, and D. Gatlin, “On Designing Universal Logic Blocks and Their Applications for FPGA Design,” *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 5, pp. 519-527, May 1997.

[17] F.P. Preparata and D.E. Muller, “Generation of Near-Optimum Universal Boolean Functions,” *J. Computer and System Sciences*, vol. 4, pp. 93-102, Apr. 1970.

[18] J. Rose, R.J. Francis, D. Lewis, and P. Chow, “Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency,” *IEEE J. Solid-State Circuits*, vol. 25, no. 5, pp. 1,217-1,225, Oct. 1990.

[19] *Representations of Discrete Functions*, T. Sasao and M. Fujita, eds. Boston: Kluwer Academic, 1996.

[20] E.M. Sentovich et al., “SIS: A System for Sequential Circuit Synthesis,” Memorandum No. UCB/ERL M92/41, Univ. of California Berkeley, May 1992.

[21] H. Stone, “Universal Logic Modules,” *Recent Developments in Switching Theory*, A. Mukhopadhyay, ed., pp. 230-254. Academic Press, 1971.

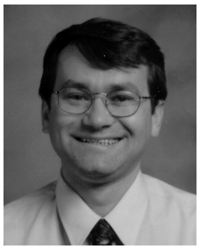
[22] E. Tau, D. Chen, I. Eslick, J. Brown, and A. DeHon, “A First Generation DPGA Implementation,” *Proc. Third Canadian Workshop Field Programmable Devices, FPD ’95*, pp. 138-143, Montreal, May 1995.

[23] S. Thakur and D.F. Wong, “On Designing ULM-Based FPGA Logic Modules,” *Proc. Third Int’l Symp. FPGAs*, pp. 3-9, Monterey, Calif., Feb. 1995.

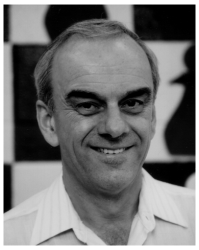
[24] S. Thakur and D.F. Wong, “Universal Logic Modules for Series-Parallel Functions,” *Proc. Fourth Int’l Symp. FPGAs*, pp. 31-37, Monterey, Calif., Feb. 1996.

[25] C.C. Tsai and M. Marek–Sadowska, “Boolean Matching Using Generalized Reed-Muller Forms,” *Proc. Design Automation Conf. ’94*, pp. 339-344, San Jose, Calif., June 1994.

[26] S. Yang and M.J. Cieselski, “Optimum and Suboptimum Algorithms for Input Encoding and Its Relationship to Logic Minimization,” *IEEE Trans. Computer–Aided Design*, vol. 10, no. 1, pp. 4-12, Jan. 1991.



Zeljko Zilic received his PhD degree from the University of Toronto, Canada, in 1997. He worked for Lucent Technologies in 1997-1998. He is now an assistant professor at McGill University, Montreal, Quebec, Canada. Dr. Zilic has been involved in the circuit design of two original local area networks, one shared-memory multiprocessor, and one FPGA architecture. He consulted for eight companies, published more than 30 papers, and has four patents pending. His current research interests include deep-submicron VLSI design methodologies, distributed shared memory machines, and interpolation algorithms.



Zvonko G. Vranesic received the BASc, MASc, and PhD degrees in electrical engineering from the University of Toronto, Canada, in 1963, 1966, and 1968, respectively. From 1963 to 1965, he worked for Northern Electric Co. Ltd., Bramalea, Ontario, Canada. In 1968, he joined the faculty of the Departments of Electrical Engineering and Computer Science at the University of Toronto, where he is now a professor. During the academic years 1977-1978 and 1984-1985, he was a senior visitor in the Computer Laboratory at the University of Cambridge, England, and in the Institut de Programmation at the University of Paris, France.

Dr. Vranesic’s research interests include computer architecture, VLSI systems, local area networks, and many-valued switching systems. He is the coauthor of three books and has published more than 100 scientific papers. He was the chairman of the Third International Symposium on Multiple-Valued Logic in 1973 and of the 18th International Symposium on Computer Architecture in 1991.