Using BDDs to Design ULMs for FPGAs

Zeljko Zilic and Zvonko G. Vranesic
University of Toronto, Department of ECE

Abstract

Many modern FPGAs use lookup table (LUT) logic blocks which can be programmed to realize any function of a fixed number of inputs. Since permutations and negation of signals are virtually costless operations in FPGAs, 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 only recently been considered for application in FPGAs. In this paper we propose a class of ULMs useful in the FPGA environment. Methodology for systematic development of such blocks is presented, based on BDD description of logic functions. We give an explicit construction of a 3-input LUT replacement that requires only 5 programming bits, which is the optimum for such ULMs. A realistic size 4-input LUT replacement is obtained which uses 13 programming bits. Such logic blocks are especially important when FPGAs are used in a reconfigurable manner, because they can reduce the time and memory needed for changing the configuration.

1 Introduction

First commercially available Field-Programmable Gate Arrays (FPGAs) had an array of 3-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 3 inputs (LUT.3). A decade or two before that, there was a significant amount of theoretical research on Universal Logic Modules (ULMs), 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. Old research on ULMs and new work on FPGAs have not been related until recently, when studies started appearing about the usefulness of ULM circuits as logic blocks in FPGAs [8], [13]. In this paper, we propose a new type of ULMs for use in FPGAs. Practical designs for 3- and 4-input LUT (LUT.3 and LUT.4) replacements are presented together with the methodology to systematically derive such blocks.

Universal Logic Modules are 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 [11]. These blocks achieve their functionality by *bridging* some inputs and/or *assigning* them to a constant, which are also assumed to be costless operations. This concept is illustrated in Figure 1.a. 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.

0

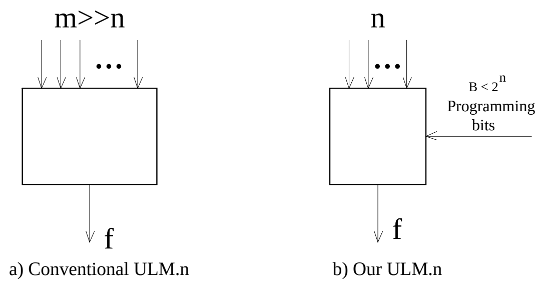


Figure 1: ULM alternatives

Recent research on ULMs has been focused on investigating the tradeoff between the functionality of logic blocks and their usefulness in real applications. In [13] it is shown that cells with up to 8-inputs can be used to compete with the Actel family of commercial FPGA circuits. The goal of that research, and the work presented in [8], is 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 [8] and LUT.4 [13], but they are not functionally complete. [1] The block in [8] has four inputs and realizes 10 out of 14 nonequivalent 3-input functions, while the block from [13] requires 8 inputs to realize almost all 4-input functions.

In this paper, we argue that a different type of ULMs is needed. Since for realization of an n input function we have to provide $m = O(2^n/log(n))$ inputs in a standard ULM, an unreasonable amount of routing resources may be needed if such blocks are used. 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 [8], the total number of inputs is limited to 4, as opposed to 8 as in [13]. We propose a class of ULM circuits that avoids this problem and limits the number of input pins to n by using separate *programming bits.* Like FPGAs, these ULMs are programmed by serial input 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. Figure 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 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 reconfigurability of FPGAs is essential [1], [7], [12]. The logic block presented in [12] contains memory storage for 4 "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 4.

Our goal is to reduce the *length of the description* of Boolean functions and to develop logic blocks which can use

[1] The block in [8] is named "semi-ULM" to express the fact that it is not functionally complete

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.

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 sufficient if the ULM circuit is too complex or too slow to be of practical use. In this paper, we design 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 operations is commonly called *npn-equivalence* [6]. 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 npn-class will be considered here primarily as a theoretical shortcut in developing the main results.

The equivalence classes of switching functions have been investigated in early studies of switching functions [5]. Using group theoretic tools, 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 taking a logarithm of Equation 1 in which the factorial function is replaced by Stirling approximation we obtain:

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

This bound rapidly approaches the size of the original lookup table 2^n. Hence, for large n, ULMs are not practical.

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 4 (as opposed to 8 in LUT.3), while for $n = 4$ and $n = 5$, the minimal number of programming bits B is 8 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 would invert the polarity of the output.

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 *representative* in literature. To generate all functions of n variables in a ULM, it is sufficient to have a block that realizes only the representatives. Furthermore, only functions that depend on exactly n variables have to be considered, because functions of fewer variables can be obtained by assigning a constant value to some of the inputs.

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) [3] to realize a complete set of representative functions. Additional switches, which select the function to be realized, are used to reconfigure the BDD structure. 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 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 circuits that use the programming bits to configure the desired function.

Based on this scheme, we design 3- and 4-input logic blocks.

3 Realization of ULM.3

It is sufficient to enumerate only the npn-equivalent functions, because 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 3 variables. Table 1 shows these classes and the number of functions they represent.

For each representative three-variable function, a BDD can have up to 5 nonterminal nodes. The union of these classes will, therefore, have at most 5 of these nodes, plus an interconnection structure and programming switches needed to program the ULM.3. All 10 representative BDDs are shown in Figure 2. The input (control) variables are ordered as:

No.	Class representative	# f's
1	$x_1x_2x_3$	8
2	$x_1x_2x_3 + \bar{x}_1\bar{x}_2\bar{x}_3$	12
3	$x_1x_2 + x_1x_3$	24
4	$x_1x_2 + \bar{x}_1\bar{x}_2x_3$	24
5	$x_1x_2\bar{x}_3 + x_1\bar{x}_2x_3 + \bar{x}_1x_2x_3$	8
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	$x_1\bar{x}_3 + x_2x_3$	24
9	$x_1x_2x_3 + x_1\bar{x}_2\bar{x}_3$	12
10	$x_1x_2 + x_1x_3 + \bar{x}_1\bar{x}_2\bar{x}_3$	24

Table 1: Equivalence classes for functions of three variables

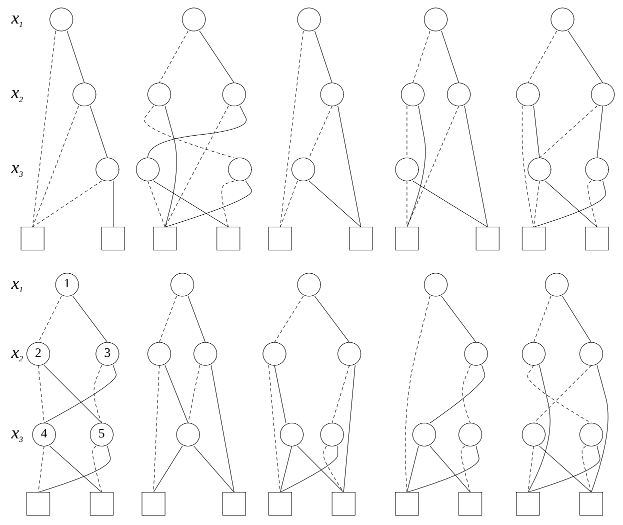


Figure 2: All representative BDDs for three-variable functions

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 each node is taken when 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.

3.1 Super BDD as a ULM

We can combine the BDDs to create the Super BDD (SBDD.3) in Figure 3 that is capable of realizing all 10 class representative functions of three variables. This union structure has 5 nonterminal nodes, which we label as in the sixth BDD in Figure 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 Figure 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. inverts the function.

The design in Figure 3 requires 6 switches, if the npn-equivalence is assumed. Seventh switch, S_7, is needed for the

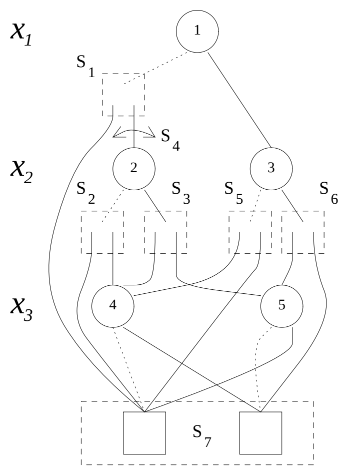


Figure 3: Super BDD (SBDD.3)

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	

Table 2: Programming bits for three variable representative functions

np-equivalence, in which case the terminal nodes can have two possible sets of values. The SBDD.3 does not implement functions of two or one variables directly; this is done 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 7, which gives us 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 more compactly all possible configurations of switches. To optimize the encoding of programming bit patterns, we enumerate all possible switch assignments for each representative function. The programming bit settings for each function are listed in Table 2. 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 2 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 the *input encoding* [15]. Since there are 10 programming combinations, 4 bits are sufficient to encode all of them uniquely. However, this alone does not lead to the simplest logic circuits. To simplify the decoding circuits, we use the functional composition and reorder some of the BDDs in Figure 2.

Switches 1 through 4 in Table 2 can be encoded separately,

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 3: Programming bits for permuted representative functions

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 4: Encoding for the first four switches

because the functions can be decomposed. Since there are 6 combinations of these switches in Table 2, three encoding bits are necessary for these four switches, and 5 bits are needed for the whole circuit. However, the optimal ULM.3 should use only 4 bits in total, as shown in Section 2.1. To obtain the optimal length encoding, we can have at most 4 programming combinations for these 4 bits. To achieve the 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 bold typeface in Table 2). Table 3 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 3 replace the corresponding functions in Table 2. With these replacements, there is a total of 4 different combinations of the programmable switches S_1 through S_4. These combinations are given in Table 4. An efficient encoding that satisfies the constraints in the table is given by:

$$S_1 = B_0 + B_1$$

$$S_2 = S_3 = B_0 \cdot B_1$$

$$S_4 = B_1$$

This allows a simple implementation of the decoder using only two 2-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 5.

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 2-input gates, while the programming bits can be kept in a standard SRAM memory. The implementation of ULM.3 can follow the layout of the SBDD.3, which has physically a fairly rectangular shape. Figure 4 compares its layout with the LUT.3 logic block. The ULM.3 switches correspond to those in SBDD.3. We targeted pass-gate CMOS implementation for most of the logic, as in many LUT implementations [10].

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	

Table 5: Optimized programming bits

	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

Table 6: Comparison between LUT.3 and ULM.3

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 [10]. Since there is a possibility of bidirectional current flow when the input to LUT changes, 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 towards 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.

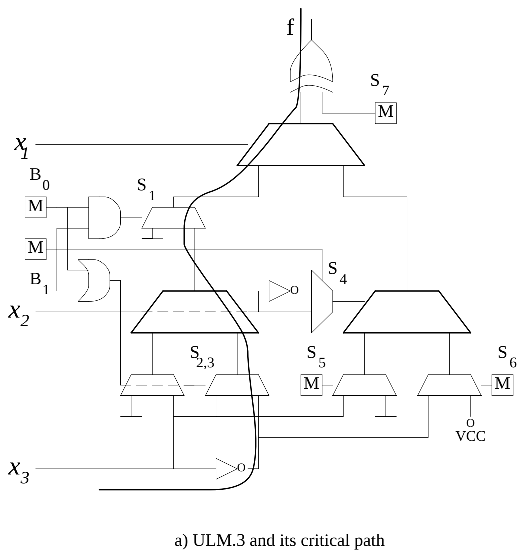
The ULM.3 implementation also has some advantages when the delay is considered. Paths of input signals can be kept shorter, which limits the propagation delay when the gate is in use. Figure 4 shows the critical paths for the two circuits, from input x_3 to output f. Speed can be improved further by placing an inversion switch in node 3, instead of node 2, of the SBDD.3.

Both LUT.3 and ULM.3 in Figure 4 have been simulated in Spice using 0.8 μm BNR BATMOS technology [2]. 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 comparison is summarized in Table 6.

One other variation is possible; the block can have the function output available in both its true and complemented form. It would be then for routing resources to select the proper polarity (or both). In this case, only 4 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 on the example of ULM.4 the procedure of (computer aided) search for an effective logic block. It is theoretically possible to derive a 4-input block that uses a minimal number of programming bits (8 for npn-equivalent



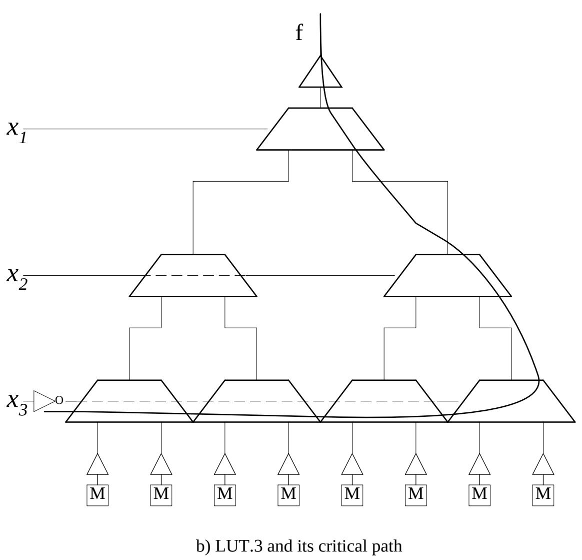


Figure 4: Implementation of ULM.3 and LUT.3

and 9 for np-equivalent ULM). However, since the minimal-length encoding may be too expensive to implement, we investigate trade-offs between the encoding length and the circuit complexity by using nonoptimal encodings as well.

There are 222 npn-equivalence classes of 4-variable functions; 208 of them depend on exactly 4 variables. These classes are enumerated in reference [4]. 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 outgoing edges from each node in the BDD. Then, a possible Super BDD structure was constructed, which had a number of programmable switches. The subsequent 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 the class representatives in [4] 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 maximal number of nodes in individual BDDs dictates how many nodes there should be in the SBDD.4. Theoretically, the largest BDD representing a 4-variable function should have at most 9 nodes, excluding 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 labelled by combining the labels of its 0- and 1-successors using the expression

$$label(v) = 2 * label(v.0) + label(v.1) \tag{2}$$

Figure 5 shows an example of the 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 $\bar{x}$, respectively.

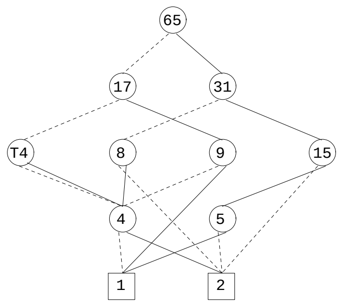


Figure 5: Example BDD labels

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 label. Number 9 can be obtained as $9 = 2 * 2 + 5 = 2 * 4 + 1$, 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 Figure 5.

To optimize the interconnection, 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 Figure 5. 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.

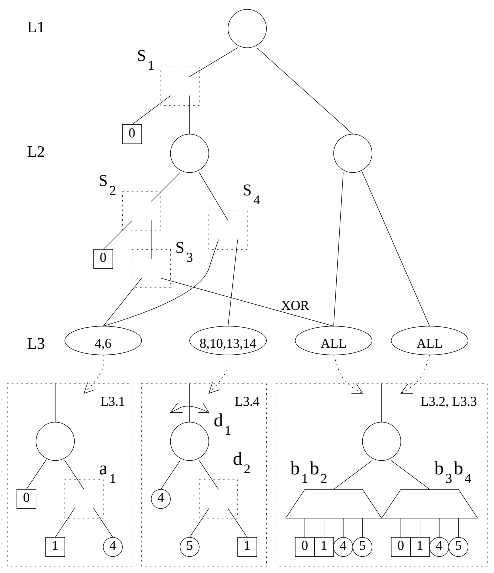


Figure 6: Optimized SBDD.4

4.2 Generating Super BDD (SBDD.4)

The number of nodes in SBDD.4 is calculated 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 SBDD.4. There are 12 possible labels for the level-3 nodes, but only 4 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.

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.

The encoding of the switches is analyzed and grouped with respect to the configuration of the first four switches. All the functions are enumerated in Table 7. 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 Figure 6. This implementation requires 15 programming switches and hence 15 programming bits.

4.3 Input encoding for programming bits

To reduce the number of programming bits, we can encode the possible switch settings. To solve this problem, we use the input encoding algorithm in NOVA, which is included in the SIS [9] package. Since the minimal-length encoding is expected to be expensive, we consider several other encoding strategies.

The four switches S_1 through S_4 can be encoded using three bits for 5 possible configurations, corresponding to groups G_1 to G_5, as shown in Table 8. This leads to a simple decoder:

$$S_1 = B_3$$

$$S_2 = B_1$$

$$S_3 = B_2 \cdot B_3$$

$$S_4 = B_2 + B_3$$

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

Table 8: Encoding for first four switches

Fourteen programming bits are needed in this arrangement. Note that group G_5 in Table 7 has exactly one function in it (the 4-input XOR), which costs a programming switch, S_3. The edge emanating from this switch is marked as "XOR" in Figure 6.

Further reduction in the number of programming bits can be achieved through more careful encoding of functions in each of the existing 5 groups. Notice that this encoding does not affect the speed of the circuit. All the 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 needing less than 10 gates for all the decoding circuitry in the ULM.4. The encoding of functions is based on sharing bits among the sub-blocks and encoding the XOR function as part of group G_4. We omit the details of this encoding for lack of space. The threshold of 10 gates is 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 expense in the hardware. To go further, to the theoretical limit of 8 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.

5 Other Issues

5.1 Technology mapping using ULMs

The ULM blocks 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 [14].

Assuming that the interconnection resources in 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 [8] found, using the MCNC benchmarks, that their ULM needs both polarities for only 6.6% of the nodes, and that the required increase in the number of blocks is 9% compared to LUT.3. Since that study was done using a block with incomplete functionality, the increase in the number of logic blocks using our block of the same granularity (ULM.3) can be in the worst case 6.6%. It is sufficient to replicate those blocks with polarity disagreement at the output. Also, since nodes tend to fanout less when granularity of the logic block increases, the results for our larger blocks (ULM.4) can only be better.

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

Table 7: Encoding groups

5.2 Functionally incomplete blocks

The motivation of work in [8] and [13] was to investigate the construction of functionally incomplete blocks. Their blocks implemented 10 out of 14 and 201 out 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 5 input pins [11]. Hence, the only reason why 4 pins were sufficient in [8] 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 3-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 Figure 6, for which there are 4 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 total decoding circuits consisting of only two 2-input gates. For comparison, the block in [13] realizes the same number of functions, but 8 input pins are required.

One more bit can be removed by excluding two more functions, f_{201} and f_{207}, but with the added price of one more 2-input decoding gate. This incomplete ULM circuit would require 11 bits for dual-output block and 12 bits for single-output np-equivalent block and it would realize 199 of 208 class representatives.

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 BDDs for both classification and realization of Boolean functions. As an illustration, we showed detailed designs of replacements for 3- and 4-input lookup tables. In the case of ULM.3, only 5 programming bits are needed, for a block slightly smaller than LUT.3. For ULM.4, several alternatives with different tradeoffs between the number of programming bits and the complexity of the circuit are considered. A circuit that requires 13 bits is devised, such that it is smaller than LUT.4. Furthermore, while the known ULM circuits considered for application in FPGAs [8], [13] are functionally incomplete, our construction offers the complete functionality at a reasonable price.

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

Acknowledgments This work is supported by Information Technology Research Centre (ITRC) of Ontario. We also thank Guy Lemieux from the University of Toronto for his constructive suggestions.

References

[1] Atmel Corporation, "Configurable Logic Design and Application Book", Atmel, San Jose, CA, 1995.

[2] Bell Northern Research, "Design Rules for CMC 0.8-micron BiCMOS, a Version of BATMOS", Ottawa, Canada, 1993.

[3] R. E. Bryant, "Graph-Based Methods for Boolean Function Manipulation", *IEEE Transactions on Computers*, Vol. 37, No. 8, Aug. 1986, pp. 677-691.

[4] 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 Transactions on Computers*, Vol. 27, No. 1, January 1979, pp. 76 – 85.

[5] M. Harrison, "Counting Theorems and their Applications to Classification of Switching Functions", *Recent Developments in Switching Theory*, edited by Amar Mukhopadhyay, Academic Press, 1971, pp. 86-121.

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

[7] D. Jones and D. Lewis, "A Time Multiplexed FPGA Architecture for Logic Emulation", *Third International Symposium on FPGAs, FPGA95*, Monterey Bay, California, February 1995, pp. 121-126.

[8] C.C. Lin, M. Marek–Sadowska and D. Gatlin, "Universal Logic Gate for FPGA Design", *Proceedings of ICCAD94*, pp. 164-168

[9] E. M. Sentovich et al., *"SIS: A System for Sequential Circuit Synthesis"*, Memorandum No. UCB/ERL M92/41, University of California Berkeley, May 1992.

[10] Soon Ong Seo, *A High Speed FPGA Using Programmable Minitiles.* M. A. Sc. Thesis, University of Toronto, 1994.

[11] Harold Stone, "Universal Logic Modules", *Recent Developments in Switching Theory*, edited by Amar Mukhopadhyay, Academic Press, 1971, pp. 230-254.

[12] E. Tau, D. Chen, I. Eslick, J. Brown and A. DeHon, "A First Generation DPGA Implementation", *Proc. 3rd Canadian Workshop on Field Programmable Devices, FPD95*, Montreal, May 1995, pp. 138-143.

[13] S. Thakur and D. F. Wong, "On Designing ULM-Based FPGA Logic Modules", *Proceedings of the Third International Symposium on FPGAs*, Monterey Bay, California, February 1995, pp. 3-9.

[14] C. C. Tsai and M. Marek–Sadowska, Boolean Matching Using Generalized Reed-Muller Forms, *Proceedings of DAC94*, pp. 339-344

[15] S. Yang and M. J. Cieselski, "Optimum and Suboptimum Algorithms for Input Encoding and Its Relationship to Logic Minimization", *IEEE Transactions on Computer–Aided Design*, Vol. 10, No. 1, January 1991, pp. 4 - 12.