Design of Arbiters and Allocators Based on Multi-Terminal BDDs

Václav Dvořák
(Brno University of Technology, Brno, Czech Republic
dvorak@fit.vutbr.cz)

Petr Mikušek
(Brno University of Technology, Brno, Czech Republic
imikusek@fit.vutbr.cz)

Abstract: Assigning one (more) shared resource(s) to several requesters is a function of arbiters (allocators). This class of decision-making modules can be implemented in a number of ways, from hardware to firmware to software. The paper presents a new computer-aided technique that can produce representations of arbiters/allocators in a form of a Multi-Terminal Binary Decision Diagram (MTBDD) with close to minimum cost and width. This diagram can then serve as a prototype for a cascade of multiple-output look-up tables (LUTs) that implements the given function, or for efficient firmware implementation. The technique makes use of iterative decomposition of integer functions of Boolean variables and a variable-ordering heuristic to order variables. The LUT cascades lead directly to the pipelined design, simplify wiring and testing and can compete with the traditional FPGA design in performance and with PLA design in chip area.

Keywords: Multi-Terminal BDDs, LUT cascades, iterative disjunctive decomposition, arbiter circuits, allocators.
Categories: B.1.1, B.1.4, B.6.1, B.6.3, C.3, C.5.4

1 Introduction

Design of digital systems with a degree of regularity in physical placement of subsystems and in their interconnection has always been a much desired goal and is even more so today. A regular logic has advantages which make it more attractive: short development time, better utilization of chip area, easy testability and easy modifications all end up in a lower cost. A one-dimensional cascade of look-up tables (LUT cells) is such a regular structure.

At present, LUTs with up to 6 binary inputs and a single binary output are common components of FPGAs. Multiple-output LUTs can be thought of as composed of single-output LUTs or can be manufactured as embedded RAMs. They may provide support for reconfigurable architectures, asynchronous cascades or clocked pipelines; speed is competitive with other FPGA designs [Nakamura, 2005], layout and wiring are very easy. The multiple-output LUT cascade is a promising reconfigurable logic device for future 45 nm and 32 nm VLSI technology [Nakamura, 2005].

Realization of every multiple-output Boolean function by a LUT cascade was proved possible [Yoeli, 1970]. However, the suggested algebraic method of synthesis was not practical, as it produced redundant cascades of the same length for the simplest functions as well as for the most complex ones, and therefore necessarily cascades too long; intuitively, for simple functions short non-redundant cascades should do.

A direct synthesis of non-redundant LUT cascades comes out easily from the known representation of multiple Boolean functions in a form of Multi-Terminal Binary Decision Diagrams (MTBDD) [Yanushkevich, 2006]. Cascaded LUTs are slices (layers) of this MTBDD. The question is how to order the variables in the diagram, because the ordering influences the size and shape of the MTBDD. Among all possible orderings of variables we should find one that produces a diagram optimal in some sense (e.g., cost, width, average path length). An optimum ordering of variables can be treated as a separate problem or it can be solved concurrently with LUT cascade synthesis by iterative decomposition [Dvořák, 2007a, b]. Sequential processing of optimized MTBDDs by means of micro-engines with multi-way branching can significantly improve firmware performance [Dvořák, 2007a].

Multiple-output Boolean functions have been more recently represented by BDD_for_CF diagrams [Matsuura, 2007]. Here the ordering of variables has to be optimized first; the top-down iterative decomposition then starts from the root and after a removal of a single variable the whole diagram has to be reconstructed. A disadvantage of this approach is a large size of BDD_for_CF diagrams, because input as well as output variables are used as decision variables.

In this paper we present a heuristic technique for bottom-up iterative decomposition of integer-valued functions starting from the leaves. The main contribution is that the bottom-up synthesis of a MTBDD/LUT cascade need not know the optimum ordering of variables, because the locally optimum order of variables is generated concurrently. The obtained MTBDDs and LUT cascades can be used in hardware, firmware and software implementation of combinational and sequential functions.

The paper is structured as follows. Basic definitions and concepts are explained in Section 2. Our heuristic approach for construction of sub-optimal MTBDDs and LUT cascades is presented in Section 3. Section 4 applies the ideas to four types of arbiters, their decomposition and implementation, whereas Section 5 is similarly devoted to allocators. Experimental results are summarized in Section 6, and commented on in the Conclusion.

2 Basic Definitions

A system of m Boolean functions of n Boolean variables,

$$f_n^{(i)} : (Z_2)^n \to Z_2\,,\ \ i = 1, 2, ..., m \tag{1}$$

will be simply referred to as a multiple-output Boolean function F_n. Instead of a full function table, we prefer to use a shorthand description of a system (1) in a form of a PLA matrix, i.e., as a set of (n+m)-tuples, called function cubes, in which the first n

components correspond to the inputs and the last *m* components to the outputs – see an example in [Tab. 1].

Symbols in the PLA matrix are interpreted the following way: each position in the input plane (x_1 to x_4 in [Tab. 1]) corresponds to an input variable where a 0 implies the corresponding input literal appears complemented in the product term, a 1 implies the input literal appears un-complemented in the product term, and "-" implies the input literal does not appear in the product term. In Espresso tool [Brzozowski, 1997], a command-line option f , d , r , fd , dr , fr , or fdr selects any combination of the ON -set (f), the OFF-set (r) or the DC-set (d) in the output format (type f is a default). We will use logical type fr for each output, so that a 1 means this product term belongs to the ON-set, a 0 means this product term belongs to the OFF-set, and a "-" means this product term has no meaning for the value of this function.

	x_1	x_2	x_3	x_4	y_1	y_2
1	0	0	-	0	1	1
2	1	0	-	0	1	0
3	-	0	0	-	1	-
4	-	-	1	1	0	-
5	-	1	1	0	0	0
6	-	1	-	1	-	1
7	0	-	0	1	1	-

Table 1: The example of multiple-output Boolean function specification by cubes (n = 4, m = 2)

	0	1	-
0	0	n.a.	0
1	n.a.	1	1
-	0	1	-

Table 2: Element-wise cube intersection

Thus, the value "-" is considered uncertain, whereas 0 and 1 are certain. An element *c* of $\{0, -, 1\}^n$ is called an input cube and element *d* of $\{0, -, 1\}^m$ is called an output cube. Function F_n is incomplete if it is defined only on set $D \subset (Z_2)^n$; $(Z_2)^n \setminus D = X$ is then the don't care set (DC-set). The elements in *X* are input vectors that for some reason cannot occur; for example, two of the input 4-tuples, 0010 and 1100, have no outputs defined, so these rows are omitted from [Tab. 1]. A function (Espresso "type fr") is completely described by providing its ON-set and OFF-set. Espresso computes the DC-set as the complement of the union of the ON-set and the OFF-set. It is an error for any min-term to belong to both the ON-set and OFF-set.

Next we will review the basic notions of cube calculus [Brzozowski, 1997].

Definition 1. Compatibility relation ~ is defined on the set {0, 1, -}:

$$0 \sim 0,\ 1 \sim 1,\ - \sim -,\ 0 \sim -,\ 1 \sim -,\ - \sim 0,\ - \sim 1,$$

but the pairs (0,1) and (1,0) are not related by ~. Compatibility relation is extended to cubes $\{0, -, 1\}^n$ denoted as $c = (c_1, c_2, \ldots, c_n)$: two cubes $c, c' \in \{0, -, 1\}^n$ are compatible,

$$c \sim c' \text{ if and only if } c_i \sim c_i' \text{ for all } i,\ 1 \le i \le n;$$

in other words, two cubes *c* and *c´* are compatible if and only if they have a non-empty common sub-cube.

The compatibility relation is reflexive and symmetric, but not transitive.

Definition 2. A binary operation intersection (product) is defined on the set {0, 1, -} in [Tab. 2]. It is not defined for pairs (0, 1) and (1, 0). The intersection can be further extended to two or more compatible cubes if it is applied element-wise.

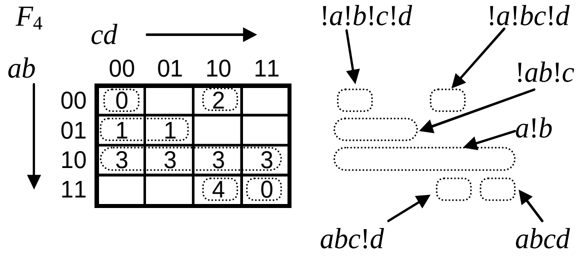
A set of (n+m)-tuples does not necessarily define a Boolean function, because it is possible to assign conflicting output values. An fr function must satisfy the consistency condition: if two input cubes are compatible, so are the corresponding output cubes [Brzozowski, 1997]. Thus if a min-term applied to the input is contained in two or more input cubes, an intersection of output cubes will be seen at the output and there will be no contradictions.

Cube specifications of general fr functions exemplified by the function in [Table 1] may contain compatible cubes and *ternary* output cubes. In this paper, we will use only a restricted class of fr functions because they are sufficient for the targeted applications and because their processing is greatly simplified. Our concern will be an incompletely specified integer (R-valued) function of n Boolean variables

$$F_n\colon D \to Z_R\,, \tag{2}$$

$D\subseteq(Z_2)^n$, $Z_R = \{0,1,2, \ldots, R-1\}$, $R \le 2^m$, such that no two input cubes are compatible. Output cubes are integer values that can be recoded back to output *binary* vectors $b \in \{0,1\}^m$ when desired. A min-term applied to the input is thus contained in one and only one input cube. Function F_n is not defined on a don´t care set $X = (Z_2)^n \setminus D$.

We will use a function F_4: $D \to Z_5$, $D\subset(Z_2)^4$ with a map at [Fig. 1] as a running example of a class of functions under our consideration. Here 6 cubes are mapped into 5 integer values. The function is not defined in $|X| = 6$ out of 16 points.



	a	b	c	d	f
0	0	0	0	0	0
1	0	1	0	-	1
2	0	0	1	0	2
3	1	0	-	-	3
4	1	1	1	0	4
5	1	1	1	1	0

Figure 1: The map of integer function F_4 and the equivalent cube specification

The most of multiple-output Boolean functions used in practice, including arbiter and allocator functions, can be expressed in cube notation with don´t cares. Cube notation is also a standard input specification for Espresso synthesis tool. We have generated cube specifications for all considered functions automatically from their definitions for any given size of synthesized arbiters and allocators.

Machine representation of single-output Boolean functions frequently uses Binary Decision Diagrams (BDDs), which can have many forms, [Yanushkevich, 2006]. Ordered BDDs (OBDDs) use the same order of variables along all paths, whereas free DDs relax this restriction. For a given variable order there exists a unique OBDD with a minimum number of decision nodes (i.e. size), so called reduced OBDD or ROBDD. The same is true for Multi-Terminal Binary Decision Diagrams (MTBDDs) representing binary-input, integer-valued output functions [Yanushkevich, 2006]. Alternatively, we can use a BDD for the characteristic function (BDD_for_CF). This kind of BDD can be obtained by existing tools [Uni. Hamburg, 2006], but ordering of variables is a separate problem, [Matsuura, 2007].

The DD size is the important parameter as it directly influences the size of the data structure needed to store the DD. However, the size of a DD is very sensitive to variable ordering and finding a good order even for BDDs is an NP-complete problem [Yanushkevich, 2006]; there are $n!$ possible orderings of n variables. The size of DDs for random functions grows exponentially with the number of variables n for any ordering, but functions used in digital system design with few exceptions do have a reasonable DD size. One exception is the class of binary multipliers: for all possible variable orderings, the BDD size is exponential for n-bit inputs and $2n$-bit output [Bryant, 1991].

We will refer to ROBDD or MTBDD with the best variable ordering as to the optimal BDD. The term a "sub-optimal DD" will denote a DD with a size near to the optimal BDD. However, in a functional decomposition, the minimization of the BDD width is more important than the minimization of the total number of nodes, because the BDD width directly influences the cascade width k. Some heuristic synthesis techniques take this into consideration [Matsuura, 2007]. The average path length (APL) of a DD that relates to the average evaluation time can also be optimized [Nagayama, 2005].

We conclude this Section by three definitions.

Definition 3. An ordered DD is non-redundant, if each test variable is used at one and only one level of the DD.

In what follows only non-redundant ordered DDs will be considered, even though redundant testing may sometimes lead to a smaller DD size.

Definition 4. A generic binary cascade C of the form $k \times 1$ is the system

$$C = [k, H_1, H_2, \ldots, H_n, \pi],$$

where

- H_i: $(Z_2)^{k_{i-1}} \times Z_2 \rightarrow (Z_2)^{k_i}$, $1 \le i \le n$ is a function implemented by the i^{th} logic device (cell) with k_{i-1} horizontal inputs, 1vertical (side) input $x_{\pi(i)}$ and k_i outputs; $(Z_2)^0 = \varnothing$.
- π is a permutation of the set $\{1, 2,\ldots, n\}$ that assigns input variable $x_{\pi(i)}$ to the i^{th} cell in the cascade, i =1, 2, …, n,
- $k = \max\,[k_i\,]$ is a cascade width,
- n, the cascade length, is the total number of cells.

Cascade cells have up to k horizontal inputs (rails) carrying Boolean values between cells and each cell has 1 additional vertical input. As first cells have $k_0 = 0$, $k_1 = 1$, $k_2 = 2$, $k_3 \in$ <1, 3>, $k_4 \in$ <1, 4>, …horizontal inputs, first t cells such that $k_t = k$ are typically combined into a single cell [Fig. 3b], so that the cascade length is then $n - t + 1$. Cell functions H_i are described by LUTs and the cascade is then referred to as the LUT cascade.

Definition 5. A cascade is said to be non-redundant if each input variable used at a vertical input enters one and only one cell. Otherwise the cascade is redundant. If a reference is made to a cascade, we will assume implicitly a non-redundant cascade.

3 Construction of LUT Cascades and of Sub-Optimal MTBDDs

In this section we present a heuristic technique for a sub-optimal LUT cascade construction. It is a generalization of the BDD construction by means of iterative disjunctive decomposition [Dvořák, 2007b]. The classical Ashenhurst-Curtis decomposition of Boolean functions works with decomposition tables constructed for various partitions of input variables $X = (X_1, X_2)$. A search for the minimum number of distinct columns in the decomposition table (so called column multiplicity) is a combinatorial problem with computation time and the memory requirements exponential in the number of inputs n. Thus the straightforward implementation of the classical method is impractical for functions with many inputs. In our approach we use partition $|X_1| = 1$, $|X_2| = i$ iteratively ($i = n$-1, n-2,…,1); in each step an input variable is selected in such a way that the width of the cascade is minimized. Simultaneously we obtain a MTBDD, which is in fact revealing the internal structure of LUTs in terms of decision nodes.

Before formulation of the algorithm, we prefer to illustrate the synthesis technique on the F_4 example [Fig. 1]. The integer function $z = F_4(a, b, c, d)$ of four binary variables is specified by cubes at the top of [Fig. 2]. In the meantime we will select the order of variables in advance as d, c, b, a. A single variable (highlighted within tables in [Fig. 2]) will be removed from the function in one decomposition step. Starting with variable d, we inspect the set of input cubes with value 0 or 1 in column d and look for all possible compatible pairs of input cubes $e = (e_1, e_2, e_3, 0)$ and $e' = (e'_1, e'_2, e'_3, 1)$ hiding their values 0 and 1. One cube (...,0) may be compatible with several cubes (...,1) and vice versa. These pairs will be referred to as binary pairs (b-pairs).

Next we will identify input cubes with value "-" in column d. From each such cube $u = (u_1, u_2, u_3, -)$ we can create a compatible pair $u = (u_1, u_2, u_3, 0)$ and $u' = (u_1, u_2, u_3, 1)$ by substitution 0 and 1 for uncertain value "-". These pairs will be referred to as unary pairs (u-pairs) because of their origin from one cube. Remaining cubes of two types, $q = (q_1, q_2, q_3, 0)$ or $r = (r_1, r_2, r_3, 1)$, are not compatible between themselves and neither with any cube in binary pairs; we will call them orphaned input cubes. This is because the compatible cubes $q = (q_1, q_2, q_3, 1)$ or $r = (r_1, r_2, r_3, 0)$ map to the DC values and therefore are not listed in the cube table. We can thus append each orphaned cube with the identical invisible input cube (denoted "x") with the DC output value. We will call these pairs appended pairs (a-pairs).

In our example in [Fig. 1] we will find

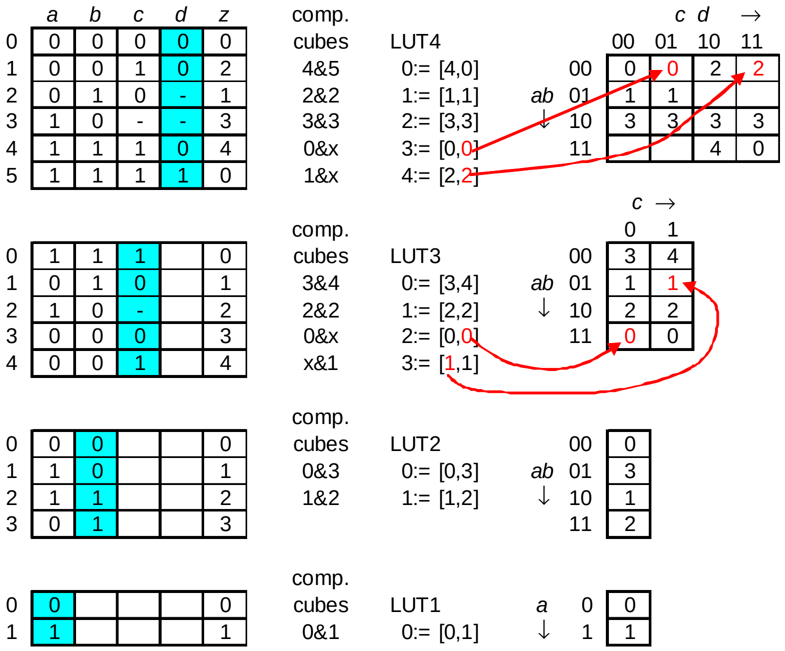
- only one b-pair, cubes 4&5
- two u-pairs, cubes 2&2 and 3&3
- two a-pairs, cubes 0&x, 1&x.

When we do decomposition of function F_4 by removal of variable d,

$$F_4 = H(G(a, b, c), d), \tag{3}$$

we have to intersect all b-, u-, and a-pairs of compatible input cubes $u = (u_1, u_2, u_3)$ and $v = (v_1, v_2, v_3)$ in order to obtain cubes of function G and map them into pairs of integer output values $[P, Q]$ as shown below:

$$\begin{array}{lrl} F_4: & u = (u_1, u_2, u_3) & F_4\,(u_1, u_2, u_3, 0) = P \\ F_4: & \underline{v = (v_1, v_2, v_3)} & \underline{F_4\,(v_1, v_2, v_3, 1) = Q} \\ G: & u * v = z = (z_1, z_2, z_3) & Z : = [P, Q] \end{array} \tag{4}$$



	a	*b*	*c*	*d*	*z*
0	0	0	0	0	0
1	0	0	1	0	2
2	0	1	0	-	1
3	1	0	-	-	3
4	1	1	1	0	4
5	1	1	1	1	0

comp. cubes	LUT4
4&5	0:= [4,0]
2&2	1:= [1,1]
3&3	2:= [3,3]
0&x	3:= [0,0]
1&x	4:= [2,2]

ab ↓ \ *c d* →	00	01	10	11
00	0	0	2	2
01	1	1		
10	3	3	3	3
11			4	0

0	1	1	1		0
1	0	1	0		1
2	1	0	-		2
3	0	0	0		3
4	0	0	1		4

comp. cubes	LUT3
3&4	0:= [3,4]
2&2	1:= [2,2]
0&x	2:= [0,0]
x&1	3:= [1,1]

ab ↓ \ *c* →	0	1
00	3	4
01	1	1
10	2	2
11	0	0

0	0	0			0
1	1	0			1
2	1	1			2
3	0	1			3

comp. cubes	LUT2
0&3	0:= [0,3]
1&2	1:= [1,2]

ab ↓	
00	0
01	3
10	1
11	2

0	0				0
1	1				1

comp. cubes	LUT1
0&1	0:= [0,1]

a ↓	
0	0
1	1

Figure 2: Iterative decomposition of an integer function of 4 binary variables

For example, pair of values [4, 0] will be produced by cubes 4 and 5 in the first table in Fig.2; without values of d are these cubes compatible and can be replaced in the new table of a residual function $G(a, b, c)$ by a single input cube 111 – their intersection. The removed variable d is left empty in all cubes of the following tables. A pair of output values [4, 0] from intersection of cubes 4&5 is replaced by a new integer id (0), as indicated in the assignment 0 := [4, 0], [Fig. 2].

Unary pairs of cubes 2&2 and 3&3 produce output pairs of the same values [1, 1] and [3, 3] redefined to new identities 1 and 2. Finally input cubes 0 and 1 are appended with the same invisible cubes to produce output pairs [0, DC] and [2, DC]. Now the DC values must be defined so as not to increase the number of existing unique pairs. If merging with one already found unique pair is not possible, like in our case, we will use pairs of the same values [0, 0] and [2, 2] and give them new identities 3 and 4; arrows in [Fig.2] show replacement of DC values in the map of F_4. Sometimes it may be useful to replace all DC values by a special default value that will be interpreted as "no _output" or "error".

Pairs of different output values correspond to a true decision node, whereas pairs of the same output values produce degenerate or false decision nodes, because variable *d* in fact does not decide anything. Nodes in the MTBDD are labeled by the new identities of output pairs. There is one true node (0) and four false nodes (1, 2, 3 and 4 shown as black dots) in the lowest level of the MTBDD in [Fig. 3a].

By now, we have exhausted all possible pairs of compatible cubes of F_4 with d = 1 and d = 0 and have replaced them by new shorter cubes of the residual function *G*. As a result of the removal of variable *d* from function F_4, each unique pair of function values can be assigned an integer id, what becomes one row of the LUT4, [Fig.2]. The number of LUT rows must be augmented by dummy rows to the nearest power of 2. The same procedure is repeated in the following decomposition steps until all variables have been removed. We proceed in a backward direction, from the leaves of the MTBDD to its root or from LUT4 to LUT1, [Fig. 3a, 3b]. In the case of LUT cascades, it is sufficient to go on with iterative decomposition only until the number of remaining variables equals the required number of binary inputs of the 1st LUT.

The remaining question not addressed as yet is, which variable should be used in any given step. We use a heuristic that strives to minimize the LUT cascade width. At each step, a variable is selected that generates the minimum number of rows in the sought LUT or equivalently the minimum number of decision nodes *w* (including *d* false nodes) in the sought level of the MTBDD. In the case of a tie, the lowest cost criterion is applied: a variable producing the lowest number $(w - d)$ of true decision nodes in the current level of the MTBDD is taken. In the case of a tie again, a variable is selected randomly.

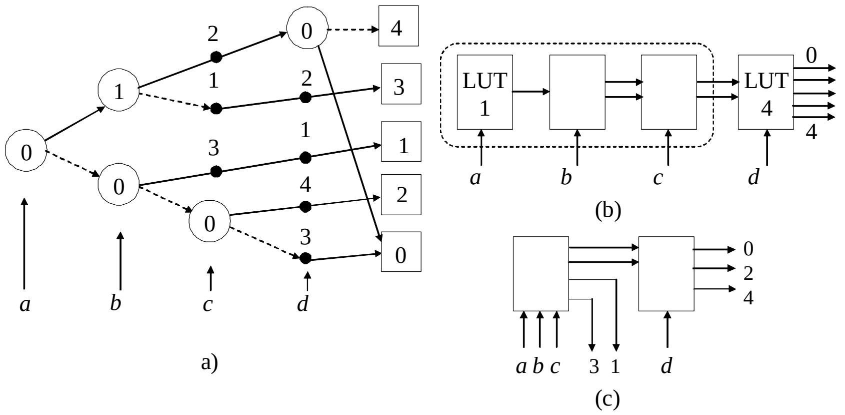


Figure 3: Iterative decomposition of the function from Fig. 3:
a) MTBDD, b) a generic LUT cascade, and c) a useful LUT cascade

The generic LUT cascade [Fig. 3b] can be shortened to two LUTs by combining first three LUTs as shown. The "1 out of 5" coding of outputs can be re-coded into a dense 3-bit code by a decoder built-in into the last LUT4, decreasing the cost of this LUT. Yet another cost-saving measure is a technique known as "intermediate outputs". As it is seen in [Fig. 3a], some terminal values (1 and 3) are generated much earlier than in the last level of the MTBDD. We can take them out from the first LUT cell in [Fig. 3c]. The cost in bits of these three LUT cascades (generic, shortened + decoder, intermediate outputs) is 66 : 50 : 56.

There is another practical reason why we need to reduce the cascade length – overall delay and the resulting speed. We can combine several consecutive LUT cells easily into one cell to reduce the delay. The cost in bits of this single cell is of course mostly larger than the aggregate cost of combined cells. Combining two cells with the same number of inputs x and outputs y does not increase the overall cost only if the single cell has one more input as against two original cells:

$$2*(y2^{x}) = y2^{x+1}. \tag{5}$$

Useful cascades shorter than n will be referred to as *reduced cascades*.

At this point it is appropriate to mention the possible shapes of MTBDDs. From the above example, it could be erroneously deducted that a profile of a MTBDD (all node count per level from leaves to the root) is always non-increasing. However, the profile generally may not be monotonic. We will consider three cases:

1. $R = |D| = 2^n$: the worst case fully specified function, the MTBDD is complete binary tree with $2^n–1$ true nodes; nodes/level drop away by one half from the previous level.
2. $R < |D| = 2^n$: fully specified functions, some function values occur more times.
3. $R \leq |D| < 2^n$: incomplete functions (with don´t cares).

Class 2 is in the worst case characterized by a profile increasing to a certain maximum and than decreasing towards the root. This is because the first level of the MTBDD can contain no more than R^2 nodes, as there are at most that many pairs of output values. Similarly the following levels cannot have more than R^4, R^8,… nodes. So the up-down profile will culminate at the point where the complete tree growing from the root meets the restriction ascending in the opposite way from leaves.

MTBDDs of incomplete functions (class 3) are limited similarly and, moreover, also by cardinality $|D|$ of the function domain: $|D|$ points cannot create more than $|D|$ compatible pairs including appended pairs. This makes the MTBDD look like a "table mountain" with southern slopes (at leaves) much steeper than northern slopes (at the root). The upper bound on the number of decision nodes was found in [Dvořák, 1997].

To aid LUT cascade synthesis, the program tool HIDET (Heuristic Iterative Decomposition Tool) has been developed. It implements the algorithm in [Fig. 4] (letters S stand for sets, M and L for tables, w for local MTBDD width (the number of all nodes) and d for the number of false nodes. The outer loop (12–34) does n steps of iterative decomposition whereas the inner loop (16–25) is looking for the best variable in each decomposition step. The inner loop tests all available variables of the yet-to-be decomposed function: selects a variable (14), initializes the local MTBDD cost measures: width *w* and *d* (15), creates cube pairs with the actual variable valued 0 and 1 (b-pairs) or the actual variable uncertain – u-pairs (17), eliminates redundant pairs (18) and merges output values of orphaned cubes with existing unique output pairs or with itself (19). Then the cost measures *w* and *d* are updated (20 - 21) and the condition is tested, whether the currently the best variable should be replaced by the actual one: if the new local width *w* is smaller than the current one, it is replaced. If it is the same, then the replacement takes place only if there are more false nodes *d* among all *w* nodes than for the best variable so far (22 – 23). When the loop terminates, the best variable is known (26).

```
1.    Input:
2.      $M_{in}$, the given table of function cubes;
3.      $S_v$, the set if input variables
4.      $n = |S_v|$,  number of input variables;
5.
6.    Output: $i$ in 1 to $n$
7.      $M_i$, a cube table of the i$^{th}$ residual function;
8.      $L_i$,  $i^{th}$  LUT counted from the end of a cascade;
9.      $v_i$, the variable removed in step $i$ ;
10.
11.   Initialize $i \leftarrow 1$, $M_0 \leftarrow M_{in}$;
12.   for $i$ in 1 to $n$ do
13.     // Determine the best variable //
14.     $v_{best} \leftarrow$ arbitrary variable from $S_v$,
15.     $w_{best} \leftarrow$ size($M_{i-1}$), $d_{best} \leftarrow 0$;
16.     for all variables $v \in S_v$ do
17.        $M_p \leftarrow$ compatible_pairs($M_{i-1}$, $v$);            [make b- and u- pairs]
18.        $S_p \leftarrow$ unique_pairs($M_p$);           [unique output pairs = LUT rows]
19.        $S_m \leftarrow$ merge_pairs($S_p$);           [a-pairs: append one output value ]
20.        $w \leftarrow$ size($S_m$);                              [all nodes, true and false]
21.        $d \leftarrow$ number of pairs of the same values in $S_m$;         [false nodes]
22.        if ($w < w_{best}$) or (($w == w_{best}$) and ($d > d_{best}$))
23.             then $v_{best} \leftarrow v$, $w_{best} \leftarrow w$, $d_{best} \leftarrow d$;
24.        endif
25.     endfor
26.   $v_i \leftarrow v_{best}$;
27.   //  Decompose //
28.   $M_p \leftarrow$ compatible_pairs($M_{i-1}$, $v_i$);
29.   $S_p \leftarrow$ unique_pairs($M_p$);
30.   $S_m \leftarrow$ merge_ pairs($S_p$);
31.   $L_i \leftarrow$ enumerate_pairs($S_m$);
32.   $M_i \leftarrow$ replace pairs in $M_p$ by new id numbers from $L_i$;
33.   $S_v \leftarrow S_v \setminus \{v_i\}$;
34.   endfor
```

Fig.4: The symbolic HEDIT algorithm for iterative decomposition

The decomposition is then repeated with the best variable (28 – 30), the output pairs are enumerated (31), the function cubes of a residual function are created (32) and the best variable is removed from the set (33). A new iteration of the outer loop then starts with the residual function short of that variable.

We have done designs with up to several hundred cubes, the largest LRS8 arbiter with 36 inputs and 8 outputs needed 1025 cubes. The sequential processing time on the Pentium-based PC has been for all presented designs between 10 ms (RRA3) and 1s (LRS6). We could not test the program on a standard benchmark set, because most of the benchmark circuits are specified by general fr functions with possible compatible input cubes and with non-empty DC set. As yet, the first version of HIDET can accept only a restricted class of fr functions as mentioned above; it is not difficult to show that the same restriction holds for all residual functions and the use of HIDET is thus correct. The next version of HIDET should address a general case of fr functions, too.

4 Arbiter Circuits

LUT cascades have been applied to many useful digital function modules [Sasao, 2005a], [Sasao, 2005b], [Sasao, 2006], [Sasao, 2007], [Qin, 2006] and their effectiveness and performance has been compared to benchmark circuits [Matsuura, 2007]. One area not addressed as yet in the context of LUT cascades is arbiter and allocator circuits. A traditional design of arbiters is discussed in [Dally, 2003]. Here we are going to apply LUT cascades to various arbiters.

We will synthesize four representative types of arbiters, namely:

1. Priority encoders with fixed priority
2. Programmable priority arbiters PPA – an arbiter with, e.g., random priority or Round Robin arbiter (RRA) with rotating priority.
3. Last Granted Lowest Priority scheme arbiter LGLP.
4. Matrix arbiter (Least Recently Served scheme, LRS).

A key property of an arbiter is its fairness. Intuitively, it is ability to provide equal service to the different requesters. For the purpose of our case study, we will use two concepts of fairness.

Definition 6. A weak fairness means that every request is *eventually* served. The maximum amount of time that a requester will wait is limited by the number of requesters.

Definition 7. Strong fairness guarantees, that requesters will be served equally often. This means that the number of times requesters are served will differ by less than ε % when averaged over a sufficient number of arbitrations.

4.1 Priority encoders

The n-input priority encoders (PEn) can serve as the simplest arbiters with fixed priority: the input request r_{n-1} has the highest fixed priority and then the priority decreases to the lowest priority level for input r_0. Output of the PEn is the address of the active request with the highest priority – the winning request. There are

$\lceil \log_2 (n+1) \rceil$ address bits since the case of no request must not coincide with any requester address. Optionally, n explicit grant outputs g_0, g_1,…, g_{n-1} can be used instead ("1-out-of n" or one-hot coding). The cube specification of the PEn consists of n+1cubes, see the example of PE4 in [Fig.5], the number of cubes is reduced by 1 in each decomposition step.

The MTBDD of priority encoders are very simple; they consist of a straight line of true decision nodes, one per each input variable (the minimum possible), and each node has another outgoing edge to a terminal node. MTBDDs of PEs with address outputs have been constructed by HIDET and their parameters are listed in [Table 4].

r_3	r_2	r_1	r_0		a_2	a_1	a_0		g_3	g_2	g_1	g_0
0	0	0	1		0	0	0		0	0	0	1
0	0	1	-		0	0	1		0	0	1	0
0	1	-	-		0	1	0		0	1	0	0
1	-	-	-		0	1	1		1	0	0	0
0	0	0	0		1	0	0		0	0	0	0

Fig. 5: Cube specification of Priority Encoder PE4

Priority encoders can be designed effectively using intermediate outputs mentioned before. To build a LUT cascade, we can create slices of the MTBDD, each slice corresponding to a single LUT. The LUT cascade has #LUT cells inter-connected by a single wire, each cell has additional n/#LUT side inputs and n/#LUT intermediate outputs, [Fig. 6].

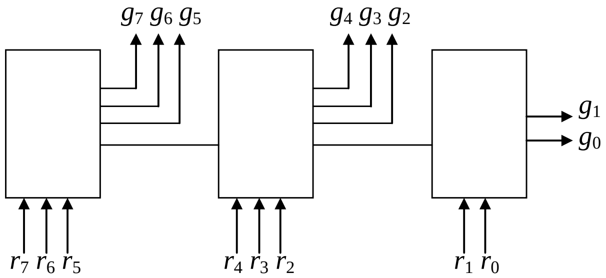


Fig. 6: The LUT cascade for PE8; n=8, #LUT=3

We can compare the LUT cascade cost (when #LUT divides n) with intermediate outputs and a cost of the PLA with address outputs. The latter is measured by the number of programmable cross-points and equals to #cubes*(2n+m); n inputs are fed into an AND array in the direct as well as in the negated form, m outputs come out of an OR array. We have:

PLA cost = *#cubes**(2n+m) = (n+1)(2n+m)
LUT cascade cost = #LUT*$2^{n/\#LUT + 1}$(n/#LUT + 1) −$2^{n/\#LUT}$ (n/#LUT + 1) − $2^{n/\#LUT + 1}$

The LUT cascade cost contains two negative terms taking into account one missing input in the first cell and one missing output in the last cell. Even though asymptotically PLA cost << LUT cascade cost, for designs useful in practice the relation is opposite:

1. $n = 8$, #LUT = 4: PLA cost = 9*(16 + 4) = 180, cascade cost = 96 – 20 = 76
2. $n = 16$, #LUT = 4: PLA cost = 17*(32 + 5) = 629, cascade cost = 640 –112= 528.

Larger designs would be composed of PEs of smaller size anyway, e.g. PE32 would be designed more efficiently as a hierarchy of four PE8 and one PE4, [Tab. 5].

For PEs it holds true that according to a subset of active requests, only a single grant output corresponding to the highest priority request in this subset is asserted. The usefulness of PEs as arbiters is in practice limited because PE is not fair, not even in the weak sense. If one request is continuously asserted, none of other requests will ever be served. There is no limit to how long a lower priority request may need to wait until it receives a grant.

4.2 A Programmable Priority Arbiter (PPA)

To improve a degree of fairness, priority of input requests must be updated dynamically. The n-bit pointer or priority register is maintained which points to the requester who is next. A single 1 in this ring register (one-hot encoding) points to the requester i, currently with the highest priority; the priority of the other inputs

$$j \bmod n,\ \ j = i-1,\ i-2,\ \ldots,\ i-(n-1)$$

decreases and is at the lowest level for the input $i–n+1 = (i+1) \bmod n$. By updating the position of 1 in the pointer register, we obtain a Programmable Priority Arbiter, PPA. Taking into account bits $p_0–p_3$ of the pointer register, we have the following conditions for asserting grant signals ($n = 4$):

$$g_3 = p_3r_3 + p_2!r_2!r_1!r_0r_3 + p_1!r_1!r_0r_3 + p_0!r_0r_3$$
$$g_2 = p_2r_2 + p_1!r_1!r_0!r_3r_2 + p_0!r_0!r_3r_2 + p_3!r_3r_2$$
$$g_1 = p_1r_1 + p_0!r_0!r_3!r_2r_1 + p_3!r_3!r_2r_1 + p_2!r_2r_1$$
$$g_0 = p_0r_0 + p_3!r_3!r_2!r_1r_0 + p_2!r_2!r_1r_0 + p_1!r_1r_0.$$

For example, if the requester 3 pointed to is active ($r_3 = 1$), it gets the grant ($g_3 = p_3r_3 = 1$). If not, the next active requester 2 gets the grant ($g_2 = p_3!r_3r_2 = 1$); if requester 2 is not active either, requester 1 gets the grant ($g_1= p_3!r_3!r_2r_1$) or the last requester 0 if it is the only active requester ($g_0 = p_3!r_3!r_2!r_1r_0$). The MTBDD for this arbiter obtained by HIDET is shown in [Fig. 7a], and the LUT cascades in [Fig. 7b, 7c]. HIDET also generated decompositions of similar arbiters with 6, 8 and 12 inputs, [Tab. 4].

There are several strategies how to program (update) the priority register of the PPA. Beside random generation of a one-hot priority vector there are three other basic methods for updating the priority vector:

1. After asserting a grant, rotate the priority vector one bit in the direction of decreasing priority (rotating Round Robin Arbiter, RRA).
2. After asserting a grant, move the 1 to the requester after the one that just received the grant (Last Granted Lowest Priority scheme, LGLP).
3. After asserting a grant, move the 1 to the first active requester after the one that just received the grant.

While a random generation of a one-hot priority vector offers only a small improvement in fairness with respect to the fixed priority arbiter, the first method is slightly better; it is considered weakly fair though, because if only a subset of inputs is active, some requests get satisfied more often than others. For example, if only requesters 3 and 0 in our 4-input RRA are active, and the pointer register points at requester 3, requester 3 will be allowed access ($g_3 = p_3r_3 = 1$) and then requester 0 will receive three grants in a row ($g_0 = p_2!r_2!r_1!r_0 = 1$, $g_0 = p_1!r_1\ r_0 = 1$, $g_0 = p_0r_0 = 1$). (Two requesters r_2 and r_1 did not use their turn with the highest priority moving in a cycle 3 → 2 → 1→ 0 → 3).

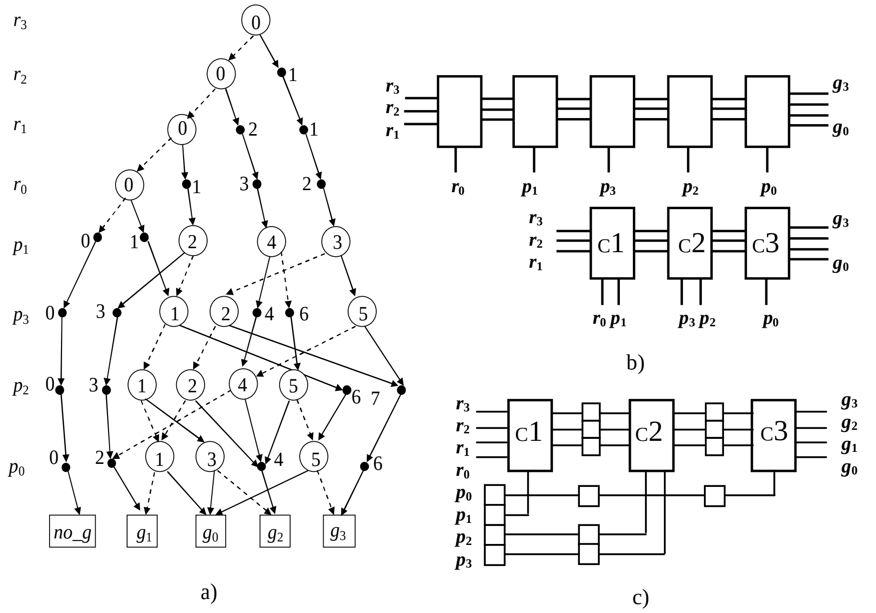


Figure 7: A MTBDD and LUT cascades for the 4-input programmable priority arbiter: a) MTBDD, b) 4- and 5- input LUTs, c) 5- input LUTs, a pipelined version.

This problem is fixed in the second method. The LGLP arbiter is almost strongly fair. Here the priority vector gets modified on the basis of the just produced grant: the Last Granted requester gets the Lowest Priority. An FSM implementation of the LGLP, that responds in one clock cycle and enables use of a resource for arbitrary number of clock cycles, will be presented below.

Finally, method three provides a strongly fair arbiter by considering only active requesters. Continuing with the above example of the 4-input RRA, when the grant is given to requester 3, method three will update the pointer register to point at the next active requester 2. After servicing requester 2, pointer register will point back at requester 3, if it remains active, etc. The kind of arbiter operating according to method three can be implemented with two RRAs working in parallel [Weber, 2001].

LUT cascades can be used for pipelined implementation of RR arbiters and for non-pipelined implementation of other types. For a pipelined implementation of the RRA, it is sufficient to complete the LUT cascade with pipeline registers between LUTs. Variables used at vertical cell inputs must also be pipelined as shown at [Fig. 7c]. The performance of the pipeline under the continuous stream of input vectors is then determined by a single LUT delay. One arbitration decision comes out every clock cycle.

4.3 The LGLP Arbiter

We will design the LGLP arbiter as a Moore type sequential state machine that responds to a request in one clock cycle and assigns the resource to a requester for one or more cycles. It has n inputs that each represents a request line, $2n$ states S_0, S_1, …, S_{2n-1} and n grant outputs. Even-numbered states monitor request inputs and odd-numbered states generate grant outputs (one grant per state). The pointer register that determines priorities of inputs is updated after servicing a request by cyclic shift in such a way that the request just satisfied gets the lowest priority. It may, but need not, stay active for the next arbitration.

Let us consider a 4-input LGLP arbiter with 8 states. Priorities of request inputs in various even-numbered states (the highest priority requests are in bold) and grant signals generated in the odd-numbered states are:

S_0:	$\boldsymbol{r_3}$	r_2	r_1	r_0	S_1:	g_3
S_2:	$\boldsymbol{r_2}$	r_1	r_0	r_3	S_3:	g_2
S_4:	$\boldsymbol{r_1}$	r_0	r_3	r_2	S_5:	g_1
S_6:	$\boldsymbol{r_0}$	r_3	r_2	r_1	S_7:	g_0

For example, in state S_0, requester 3 has the highest priority and requester 0 the lowest one. If the requester 3 is active, the state of the LGLP moves from S_0 into S_1 and grant g_3 is asserted. If requester 3 is not active, the first active requester from 2, 1 or 0 (in this order) will produce the state change into S_3, S_5 or S_7 and will get the grant g_2, g_1 or g_0 only once, unlike to the RRA behavior. State transitions for even-numbered states are thus easy to specify; e.g. for state S_0 we have:

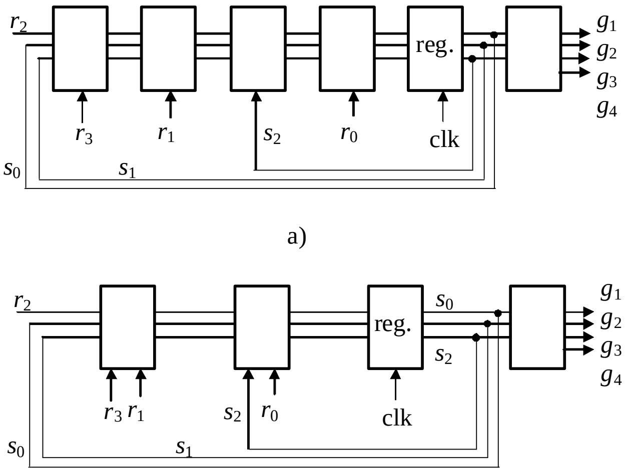
old state $s_2s_1s_0$	r_3	r_2	r_1	r_0	new state $s_2s_1s_0$
000 (S_0)	1	x	x	x	001(S_1)
000 (S_0)	0	1	x	x	011(S_3)
000 (S_0)	0	0	1	x	101(S_5)
000 (S_0)	0	0	0	1	111(S_7)

An odd-numbered state S_{2i+1}, i = 0–3, issuing the grant, transits to the next even-numbered state $S_{(2i+2)\mathrm{mod}\ 2n}$ as soon as the request terminates (goes low).

Function tables of LGLP arbiters have been generated automatically for 3, 4, 6, 8 and 12 request inputs and decomposed as before. [Fig.8a] shows for example the LUT cascade implementation with 4 LUTs obtained by HIDET. The state register clock cycle is dominantly determined by the delay of 4 LUTcells. If we combine two adjacent cells together [Fig. 8b], we can almost double the speed, provided that the delay of 4-input and 5-input cells is the same. The last cell in both cascades only transforms the state code ($s_2s_1s_0$) of odd-numbered states into grant signals g_1 to g_4. For comparison, VHDL synthesis tool for Xilinx FPGA generates this arbiter with 17 4-input LUTs in 4 logic levels.

4.4 The Matrix Arbiter with the Least Recently Serviced (LRS) Strategy

The LRS strategy cannot be implemented by a dynamically changing priority vector only; it has to use priority matrix P, where $p_{ik} = 1$ means that the i^{th} request has priority over the k^{th} request. The just asserted grant output g_j resets the j^{th} row of P to all zeros and sets the j^{th} column of P to all ones. Thus the request r_j will have priority over no other requests and all requests will have priority over r_j.



a)

b)

Figure 8: LUT cascade implementation of the 4-input LGLP arbiter.
a) 4-input LUTs b) 5-input LUTs

The priority matrix P initialized and updated according to above rules is anti-symmetric: elements p_{ik} under the main diagonal are complements of elements p_{ki} above it. It is therefore sufficient to store only elements of P above the main diagonal. For n requests we will thus need $(n^2 - n)/2$ state variables (one half of all non-diagonal elements).

As an example, we will use arbiter LRS4 that is implemented in matrix form in [Dally, 2003]. It has 6 state variables s_5, s_4, s_3, s_2, s_1, s_0 and 4 request inputs r_3, r_2, r_1, r_0. We will let the LUT cascade generate 4 grant outputs g_3, g_2, g_1, g_0, which will be then used to reset and set selectively 6 state flip-flops. This time we will implement the LRS4 arbiter in firmware. We have used again the HIDET tool to decompose function LRS4: $(Z_2)^{10} \rightarrow Z_4$. The result is shown as the MTBDD at [Fig. 9].

Evaluation of Boolean functions at the firmware level can use the MTBDD split into 3 blocks [Fig. 9]. By making use of a hardware micro-engine with a support for multi-way branching, we can speed up evaluation of Boolean functions with respect to a general purpose CPU core. A suitable architecture of a micro-engine, a modified version of the one in [Dvořák, 2007a], is depicted in [Fig. 10]. The micro-instruction format (μIF) determines the way of selecting the address of the next microinstruction.

Out of all microinstructions formats supported by the micro-engine architecture, two formats are essential for fast evaluation of multiple-output Boolean functions:

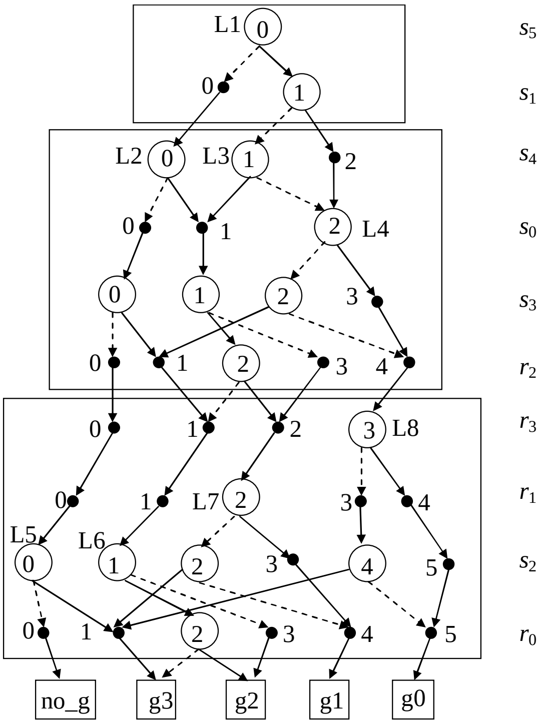


Figure 9: MTBDD of the 4-input LRS arbiter.

1. the jump to an address specified in micro-instruction modified by BCU;

```
Ln: exit Lm@x1...xk;
```

2. a conditional output (on state transition) and the jump to an address specified in micro-instruction (no modification),

```
Ln: c_output exit Lm.
```

The first format provides support for multi-way branching. It is the jump to the target address obtained from the address specified in the micro-instruction and modified by up to 4 external variables at a time, including 0 variable (no modification), by means of 16-way Branch Control Unit (BCU). Input variables are selected by multiplexers, so that a microinstruction contains MXs control field and a BCU mask.

The task of the BCU (such as Am 29803A) is to shift active inputs, selected by a 4-bit mask, to the lowest positions of the 4-bit BCU output vector. The BCU output vector is then wire-ORed with the address obtained from the microinstruction. This way we can store dispatch tables in compact form in control memory. Replacement of up to 4 bits in the address is denoted by operator "@". If wired-OR is used for replacement, the bits being replaced must be reset to 0.

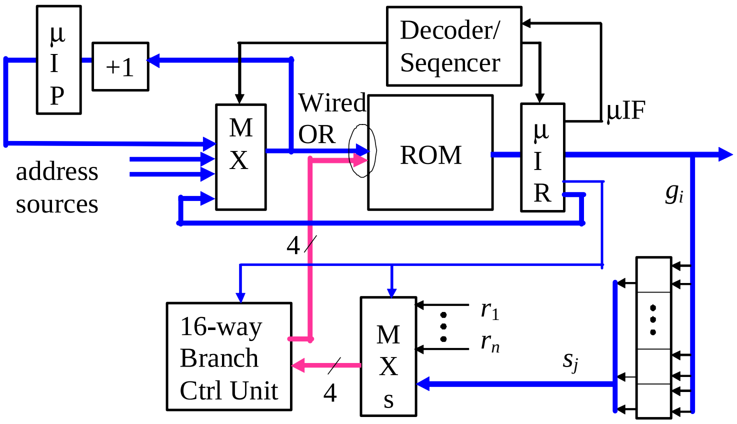


Figure 10: Micro-programmed controller architecture with multi-way branching

```
LRS: exit L1@s5s1
L1@00: exit L2@s4s0s3r2
L1@01: exit L2@s4s0s3r2
L1@10: exit L3@s4s0s3r2
L1@11: exit L4@s0s3r2
L2@0000: exit L5@s2r0
L2@0001: exit L5@s2r0
L2@0010: exit L6@s2r0
...
...
L8@1110: g0 exit Next
L8@1111: g0 exit Next
Next:
```

Figure 11: A symbolic micro-program for the LRS4 arbiter

If there are more than 4 external variables, we traverse a MTBDD in several steps testing up to 4 variables at a time. We will illustrate rewriting a MTBDD at [Fig. 9] into the micro-program with multi-way branching. The symbolic micro-program targeted for the micro-engine in [Fig. 10] is shown in [Fig. 11]. The micro-program is composed of 8 dispatch tables starting at symbolic addresses L1 to L8, [Fig. 9]. The size of dispatch tables varies according to false nodes on the path from the block inputs down to the block outputs. There will be three dispatch table of size 4 (L1, L5, L6), two of size 8 (L4, L7) and 3 of size 16 (L2, L3, L8). The total number of micro-instructions is thus

$$3\times4 + 2\times8 + 3\times16 = 76$$

and an arbitration decision is produced after execution of four microinstructions. The state of the arbiter is kept in 6 R-S flip-flops and these flip-flops are selectively set and reset by signals g_i according to the rules for updating matrix P:

$$R_0 = R_1 = R_2 = g_0,\ \ R_3 = R_4 = g_1,\ \ R_5 = g_2,$$
$$S_0 = g_1,\ \ S_1 = S_3 = g_2,\ \ S_2 = S_4 = S_5 = g_3.$$

Out of 6 state variables and 4 input requests up to 4 signal lines are selected by 4 multiplexers, fed into BCU and used in the least significant positions for address modification, as shown in [Fig. 10].

Had we used only single variable tests (a binary program with 2-way branching), we would need 16 dispatch tables of size 2, i.e. 32 microinstructions in total. However, the performance would be almost 3-times lower due to serial execution of 11 microinstructions, one in each level of the MTBDD.

5 Allocators

An $m \times n$ allocator is a unit that accepts m requests on its inputs for n distinct resources and generates grants on its outputs. It is therefore more general than an arbiter that makes decision regarding only a single resource. Any particular problem and its solution can be represented in terms of two binary-valued $m \times n$ matrices, a request matrix R and a grant matrix G. One requester may ask one or more resources, but

1. at most one grant for each input (requester) may be asserted;
2. also, at most one grant for each output (resource) can be asserted.

By means of a bipartite graph representation, the allocation can be formulated as bipartite matching problem and solved exactly, yielding the maximum possible number of assignments (*maximum* matching). *Maximal* matchings are those where no additional requests can be serviced without removing one of the existing grants.

As the exact solution of the allocation problem is too time consuming, we usually put up with approximate solution in hardware. So called requesters-first separable allocators use a set of priority encoders to select one resource per requester and then priority encoders in the second set to solve simultaneous requests for the same resource. Resources-first separable allocators do these decisions in the opposite order. Some allocators (PIM, iSLIP) reach a decision in several iterations.

Separable allocators can easily be constructed from PPAs or other arbiters. The example of 4 × 3 resources-first allocator is in [Fig. 12]. There are four requesters and 3 resources. The first index denotes a requester, the second index is the id of a resource. For simplicity, priority inputs are omitted in [Fig. 12].

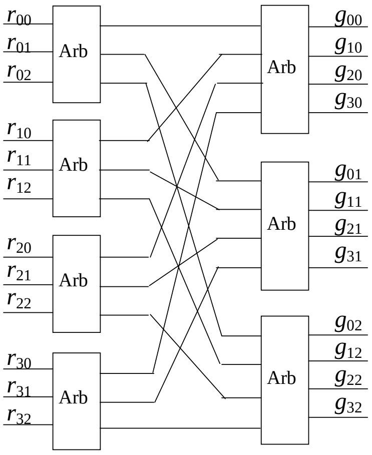


Figure 12: 4 × 3 resources-first allocator

Next we will design an $n \times n$ wavefront allocator (WFA) that arbitrates among requests for inputs and outputs simultaneously, [Fig. 13].

- It works by granting row and column tokens to a diagonal group of cells, in effect giving this group priority.
- A diagonal group k contains cells x_{ij} such that

$$(i + j) \bmod n = k.$$

- Example [Fig. 13]: Let the priority be given at the start to a diagonal group $(i + j) \bmod 4 = 3$. This is done by asserting signal p_3. Element 21 gets the grant, remaining elements send tokens down and to the right.
- An element will get a grant if it received both row and column tokens. If only one token comes or if the element has no request, tokens will pass through it. In the example, highlighted grants g_{21}, g_{00}, g_{32} will be issued in response to 8 active requests shown.
- In the next round, the following diagonal group is selected. This ensures fairness of allocations.

For example grant g_{00} is asserted under the following conditions:

$$\begin{aligned} g_{00} = {} & (p_0!p_1!p_2!p_3r_{00}) + \\ & + (!p_0\ !p_1!p_2p_3r_{00}!r_{03}!r_{30}) + \\ & + (!p_0!p_1p_2!p_3r_{00}!r_{02}!r_{03}!r_{20}!r_{30}) + \\ & + (!p_0p_1!p_2!p_3r_{00}!r_{01}!r_{02}!r_{03}!r_{10}!r_{20}!r_{30}); \end{aligned}$$

similar equations can be written for remaining 15 grant signals, 16 Boolean functions of 20 variables in total. Let us note that typically functions g_{ix} do not depend on 2-3 input variables.

To simplify the problem, the iterative decomposition has been done on four groups of outputs $g_{0x} - g_{3x}$. Output grouping has been used already in [Matsuura, 2007], where the problem of optimum grouping has been analyzed and a solution suggested. In our case grants related to one requester are defined by incompatible cubes and, due to restrictions in the present version of HIDET, have been naturally included in a single group.

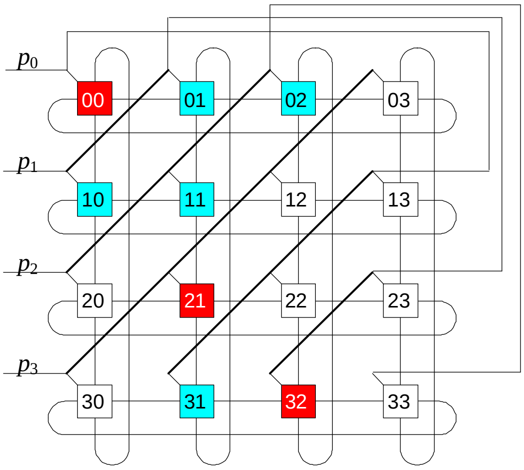


Figure 13: 4 × 4 wavefront allocator

The PLA-matrix description of four grants g_{0x} is shown in [Tab. 3]. From the number of uncertain values in 16 cubes we can figure out that the function has 67% of don't care values. The generic iterative decomposition using HIDET tool gave us the following optimal (in terms of our heuristics) ordering of variables:

```
p0, r00, p1, p2, p3, r01, r02, r11, r12, r13, r10, r20, r03,
r21, r22, r23, r30, r31, r33, r32.
```

In each decomposition step we obtained the number of true and false decision nodes in the current level of the MTBDD and the size of the associated LUT. By combining adjacent generic LUTs together one can create larger LUTs as shown in [Fig. 14].

6 Experimental results

We have generated function tables of many instances of four types of arbiters and two WF allocators automatically by means of small routines in C which enable scaling to the desired size. The example of the resulting PLA matrix description is shown at [Tab. 3]. The number of inputs, outputs and generated cubes in selected designs and processed later by the HIDET tool are given in first three columns of [Tab. 4]. Iterative decomposition of this class of functions was a matter of seconds on the Pentium-powered PC. The sample results for RRA, LGLP and LRS arbiters and WFA allocators are summarized in [Tab. 4], namely MTBDD nodes, PLA cost and cost of LUTs in generic cascades.

```
.i 20
.o 4
.ilb p0 p1 p2 p3 r00 r01 r02 r03 r10 r11 r12 r13 r20 r21 r22 r23
r30 r31 r32 r33
.ob g00 g01 g02 g03
.type fr
.p 16
10001--------------- 1000
00011--0--------0--- 1000
00101-00----0---0--- 1000
010010000---0---0--- 1000
0100-1-------------- 0100
100001-----------0-- 0100
000101-0-----0---0-- 0100
00100100-0---0---0-- 0100
0010--1------------- 0010
0100-01-----------0- 0010
1000001-------0---0- 0010
00010010--0---0---0- 0010
0001---1------------ 0001
0010--01-----------0 0001
0100-001-------0---0 0001
10000001---0---0---0 0001
.e
```

Table 3: Sample cube specification of the 4 × 4 wavefront allocator, group $g_{00} - g_{03}$

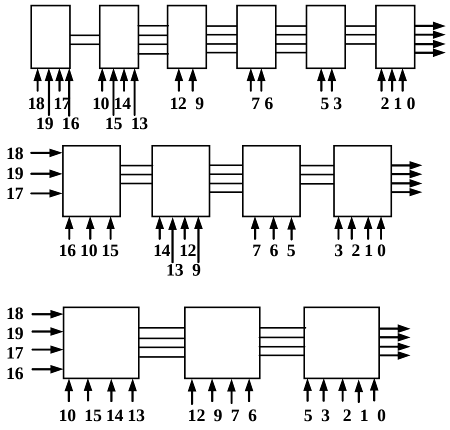


Fig. 14. The wavefront allocator - shortened cascades for group g_{0x} with 6-, 7-, and 8-input cells ($p_0 \equiv 0, \ldots, r_{33} \equiv 19$)

The last column of [Tab. 4] shows how PLA cost1 (in crosspoints) compares to the generic cascade cost2 (in bits). If we take one crosspoint in a PLA (1 gate) as expensive as 1 bit in a LUT (one storage cell), we find that cost of generic cascades is always less than that of PLA, including PEs with intermediate outputs. It seems that the only expensive designs are allocators, because PLA cost1< LUT cascade cost2. To get better results for allocators, we can decompose the problem in a better way than how it was done with partitioning outputs into groups. The wavefront allocator is in fact a 2D-RRA arbiter (requesters and resources are two dimensions) and instead of doing both dimensions simultaneously, we can do one after another: one request is selected in each row and then all selected requests for the same resource are arbitrated again. Thus the same scheme as in Fig.12 can be used for wf4 with 4 + 4 four-input RRA arbiters in LUT-cascade form; this variant is denoted as wf4+. Priority inputs are rotated one bit from every arbiter to the next one, in both groups. The cost2 of this solution is 8-times the cost1 of RRA4. It turns out that it is less than cost of a single PLA for wf4+, see column cost1 in [Tab.4].) Similar conclusions hold true for wf3 and wf3+, too; allocator wf3+ is composed of 3 + 3 three-input RRA arbiters.

[Tab. 5] summarizes the results for FPGA designs and reduced LUT cascades. We have converted PLA specification into VHDL (by means of PLA → VHDL converter ver.1.02 from Warsaw Military University of Technology) and used Xilinx FPGA synthesis tool to obtain designs based on 4-input, single output LUTs. Examples given under reduced LUT cascades are selected in such a way that LUTs are as much uniform in size as possible. LUT inputs, outputs and maximum width enable complete reconstruction of cascades. The number of LUTs and their capacity in bits are given in last two columns of [Tab. 5].

Generally, LUT cascades require larger aggregate bit capacity of LUTs than 4-input LUTs in FPGA design because of their coarser granularity, but this may be compensated by much lower wiring area of cascaded LUT cells. If we take the delay of FPGA’s 4-input LUTs plus wiring delay approximately equal to cascaded LUTs’ delay, we can compare logic levels of FPGA with cascade length (#LUTs). This way, it comes out from [Tab. 5] that we should be able to get the same or better performance with reduced LUT cascades. Of course, in the case of wf3+ and wf4+ the delay given by the number of LUTs between inputs and outputs differs from the total number of LUTs. Both the values are separated by the slash "/", see [Tab. 5].

7 Conclusions

The presented method of MTBDD/LUT cascade synthesis of multiple-output Boolean functions aided by the HIDET tool proved to be suitable for synthesis of combinational and sequential designs. The experimental results show that LUT cascades offer smaller number of logic levels than FPGA and smaller or comparable chip area than PLA (provided that a programmable cross-point and a bit of RAM occupy the same area). The method is applicable to incompletely specified, multiple-output integer functions of Boolean variables and was illustrated on the class of arbiter and allocator networks. A delay from input requests to an arbitration decision

	n	m	#cub	MTBDD	nodes		PLA	LUT	
				all	false	true	cost1	cost2	cost2/
				Σw	Σd	cost	x-points	bits	cost1
RRA3	6	3	10	20	10	10	150	146	0,97
RRA4	8	4	17	37	20	17	340	258	0,76
RRA6	12	6	37	91	51	40	1110	914	0,82
RRA8	16	8	65	179	105	75	2600	2338	0,90
RRA12	24	12	145	489	300	189	8700	8418	0,97
LGLP3	6	3	18	30	10	20	270	242	0,90
LGLP4	7	3	28	48	18	30	476	450	0,95
LGLP6	10	4	54	101	41	60	1296	1218	0,94
LGLP8	12	4	88	153	78	75	2464	1778	0,72
LRS3	6	3	13	18	8	10	195	114	0,58
LRS4	10	4	33	39	22	17	792	282	0,36
LRS6	21	6	193	111	78	33	9264	986	0,11
WF3 g_{0x}	10	3	10	45	28	17	230	274	1,19
WF3 g_{1x}	11	3	10	45	26	19	250	306	1,22
WF3 g_{2x}	10	3	10	45	28	17	230	274	1,19
WF4 g_{0x}	19	4	16	109	74	35	714	978	1,37
WF4 g_{1x}	19	4	17	108	67	41	714	866	1,21
WF4 g_{2x}	18	4	17	102	63	39	680	546	0,80
WF4 g_{3x}	19	4	17	117	79	38	714	978	1,37
PE4	4	3	5	4	0	4	55	24	0,44
PE8	8	4	9	8	0	8	180	56	0,31
PE12	12	5	13	12	0	12	377	88	0,23
PE16	16	5	17	16	0	16	629	120	0,19
PE32	32	6	33	32	0	32	2310	248	0,11
WF3+	12	9	30	not	looked	for	990	864	0,87
WF4+	20	16	68	not	looked	for	3808	2048	0,54

1. PEs are implemented with intermediate outputs, PE32 as 4 x PE8 + 1x PE4
2. WF3+ and WF4+ are implemented as 6 x RRA3 and 8 x RRA4
3. LUTcost2 in bits is the aggregated capacity of LUTs in generic cascades

Table 4: Properties of MTBDDs, PLAs and generic LUT cascade design

is given by the adjustable number of LUTs in the cascade and has been chosen less or comparable to FPGA logic levels. If the high-speed arbitration is required on a stream of input vectors, pipelining can be used. Arbitration results will then follow one after another separated by a single LUT delayThe effectiveness of LUT cascades has been proved in terms of the relative size of a PLA and the aggregate bit capacity of all LUTs in a cascade. Arbiters, as well as other digital systems frequently used in practice, have relatively low complexity,

			Reduced LUT cascades				
	FPGA		cell	cell	max.	#LUTs	Σbits
	#4-LUT	levels	inputs	outputs	width		
RRA3	9	2	5,4	3,3	3	2	144
RRA4	14	4	5,5,4	3,3,4	3	3	256
RRA6	29	5	6,6,6,5	3,4,4,6	4	4	896
RRA8	43	7	4×7,5	3,5,5,4,8	5	5	2432
RRA12	82	12	6×8,7	4,4,3×6,5,12	6	7	9472
LGLP3	10	3	4,5	3,3	4	2	144
LGLP4	17	4	5,5,5	4,4,3	4	3	352
LGLP6	48	6	6,6,6,6	4,5,5,4	5	4	1152
LGLP8	70	9	7,7,7	4,5,4	5	3	1664
LRS3	6	2	4,4	2,3	3	2	80
LRS4	8	2	6,6	2,4	3	2	384
LRS6	24	3	9,9,9	3,3,6	4	3	6144
WF3 g_{0x}	9	3	5,5,5	2,3,3	3	3	256
WF4 g_{0x}	24	4	6,7,7,7	3,4,3,4	4	4	1600
WF3+	54	4	3×RRA3	+3×RRA3	3	4/Σ12	864
WF4+	112	8	4×RRA4	+4×RRA4	3	6/Σ24	2048
PE4	3	1	2,3	3,2	1	1	48
PE8	8	2	3,4,3	4,4,2	1	3	112
PE12	15	4	4,5,5	5,5,4	1	3	368
PE16	20	5	4,5,5,5	5,5,5,4	1	4	528
PE32	40	10	4×PE8	+1×PE4	1	4/Σ13	476

Column #LUTs gives #LUTs from input to output /total number of LUTs

Table 5: FPGA designs and the reduced LUT cascade implementations

which makes their cost-effective cascade implementations possible. Beside easy interconnection there are other advantages of cascade implementation. Testing of LUT cascades reduces to a problem of testing RAM modules. Fault tolerance techniques for memories are thus also applicable for LUTs. Due to a highly developed memory technology the power consumption is very low for RAMs and it only remains to verify experimentally real power savings for specific applications.

Future research will address more general multiple-output Boolean functions specified by the Espresso fr type – those with overlapping input cubes and with ternary output cubes as in [Table 1]. That will make possible to compare the quality of our variable ordering heuristic with other heuristic methods [Yanushkevich, 2006], [Drechsler, 1998]. A redundant (non-disjunctive) decomposition or multi-variable decomposition in a single step are also of interest and could provide appropriate design techniques for new classes of functions. At present, the MTBDDs constructed by the presented technique are being applied to optimization of digital control firmware and software

Acknowledgements

This research has been carried out under the financial support of the research grants “Natural Computing on Unconventional Platforms”, GP103/10/1517, “Safety and security of networked embedded system applications”, GA102/08/1429, "Mathematical and Engineering Approaches to Developing Reliable and Secure Concurrent and Distributed Computer Systems" GA 102/09/H042 , all care of Grant Agency of Czech Republic, and “Security-Oriented Research in Information Technology”, MSM 0021630528.

References

[Bryant, 1991] Bryant, R. E.: “On the complexity of VLSI implementations and graph representations of Boolean functions with applications to integer multiplication”; IEEE Transactions on Computers, 40, (1991), 205–213,.

[Brzozowski, 1997] Brzozowski, J. A., Luba, T.: “Decomposition of Boolean Functions Specified by Cubes”; Res. report CS-97-01, University of Waterloo, Canada (1997).

[Dally, 2003] Dally, W. J., Towles, B.: “Principles and Practices of Interconnection Networks”. Morgan Kaufmann Publishers / Elsevier, San Francisco (2003).

[Drechsler, 1998] Drechsler,R., Becker, B.: “Binary Decision Diagrams - Theory and Imoplementation”. Kluwer Academic Publishers, Boston (1998).

[Dvořák, 2007a] Dvořák, V.: “LUT Cascade-Based Architectures for High Productivity Embedded Systems”; International Review on Computers and Software, 2, 4 (2007), 357–365.

[Dvořák, 2007b] Dvořák, V.: “Efficient Evaluation of Multiple-Output Boolean Functions in Embedded Software or Firmware”; Journal of Software, 2, 5 (2007), 52–63.

[Dvořák, 1997] Dvořák, V.: “Bounds on the Size of Decision Diagrams”. J.UCS (Journal of Universal Computer Science), 3,1 (1997), 2 - 22.

[Matsuura, 2007] Matsuura, M., Sasao, T.: “BDD representation for incompletely specified multiple-output logic functions and its application to the design of LUT cascades”; IEICE Transaction on Fundamentals of Electronics, Communications and Computer Sciences, E90-A, 12, (2007), 2770–2777.

[Nakamura, 2005] Nakamura, K., Sasao, T., Matsuura, M., Tanaka, K., Yoshizumi, K., Qin, H., Iguchi, Y.: “Programmable logic device with an 8-stage cascade of 64K-bit asynchronous SRAMs”; Cool Chips VIII, IEEE Symposium on Low-Power and High-Speed Chips, IEEE Press (2005).

[Nagayama, 2005] S. Nagayama, A. Mishchenko, T. Sasao, and Jon T. Butler, "Exact and heuristic minimization of the average path length in decision diagrams"; Journal of Multiple-Valued Logic and Soft Computing, 11, 5-6, (2005), 437-465.

[Qin, 2006] H. Qin and T. Sasao, "Design of address generators using multiple LUT cascade on FPGA"; Proc. SASIMI Workshop, (2006), 146-152.

[Sasao, 2005a] T. Sasao,"Radix converters: Complexity and implementation by LUT cascades"; Proc. ISMVL (2005), 256-263

[Sasao, 2005b] T. Sasao, Y. Iguchi, T. Suzuki, "On LUT cascade realizations of FIR filters"; DSD 2005 (8th Euromicro Conference on Digital System Design: Architectures, Methods and Tools), 2005, 467-474.

[Sasao, 2006] T. Sasao and J. T. Butler, "Implementation of multiple-valued CAM functions by LUT cascades"; Proc. ISMVL (2006); also available in electronic version at http://www.lsi-cad.com/sasao/Papers/pub2006.html

[Sasao, 2007] T. Sasao, S. Nagayama and J. T. Butler, "Numerical function generators using LUT cascades"; IEEE Transactions on Computers, 56, 6 (2007), 826-838.

[Uni. Hamburg, 2006] http://tams-www.informatik.uni-hamburg.de/applets/

[Weber, 2001] M. Weber: “Arbiters: Design Ideas and Coding Styles”; Proc. of the SNUG 2001 (Synopsis User Group Conference), Boston, USA (2001), 1-22.

[Yanushkevich, 2006] Yanushkevich, S. N., Miller, D. M., Shmerko, V.P., Stankovic, R. S.: “Decision Diagram Techniques for Micro- and Nanoelectric Design. CRC Press, Taylor & Francis Group, Boca Raton, FL (2006).

[Yoeli, 1970] Yoeli, M.: “The Synthesis of Multivalued Cellular Cascades”; IEEE Trans. On Computers, C-9 (1970), 1089–1090.