On Lookup Table Cascade-Based Realizations of Arbiters

Petr Mikušek, Václav Dvořák
Faculty of Information Technology, Brno University of Technology, CZ
{imikusek, dvorak}@fit.vutbr.cz

Abstract

This paper presents a new algorithm of iterative decomposition for multiple-output Boolean functions with an embedded heuristics to order variables. The algorithm produces a cascade of look-up tables (LUTs) that implements the given function and simultaneously a sub-optimal Multi-Terminal Binary Decision Diagram (MTBDD). The LUT cascade can be used for pipelined processing on FPGAs with BRAMs or at a non-traditional synthesis of large combinational and sequential circuits. On the other hand, suboptimal MTBBDs can serve as prototypes for efficient firmware implementation, especially when a micro-programmed controller that firmware runs on supports multi-way branching. A novel technique is illustrated on practical examples of three types of arbiters. It may be quite useful as a more flexible alternative implementation of digital systems with increased testability and improved manufacturability.

Keywords: *LUT cascades, Multi-Terminal BDDs, iterative disjunctive decomposition, arbiter circuits*

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 at present. 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.

LUTs are in fact multiple-input, multiple-output universal logic blocks. LUTs in block RAMs may provide support for reconfigurable architectures, asynchronous cascades or clocked pipelines; speed is competitive with other FPGA designs [1], layout and wiring are very easy. The LUT cascade is a promising reconfigurable logic device for future sub-100nm LSI technology [1]. Sequential processing of LUT cascades by means of micro-engines with multi-way branching can improve firmware performance a great deal [2].

Realization of every multiple-output Boolean function (or equivalently of an integer function of Boolean variables) by a LUT cascade was proved possible long time ago [3]. However, the algebraic method of synthesis suggested then 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.

A direct synthesis of non-redundant LUT cascades comes out easily from the known representation of integer functions of Boolean variables in a form of Multi-Terminal Binary Decision Diagrams (MTBDD), [5]. Cascaded LUTs are obtained as slices (layers) of this MTBDD. The question is how to order the variables in the diagram, because the ordering influences its size and shape. 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 [4], [2].

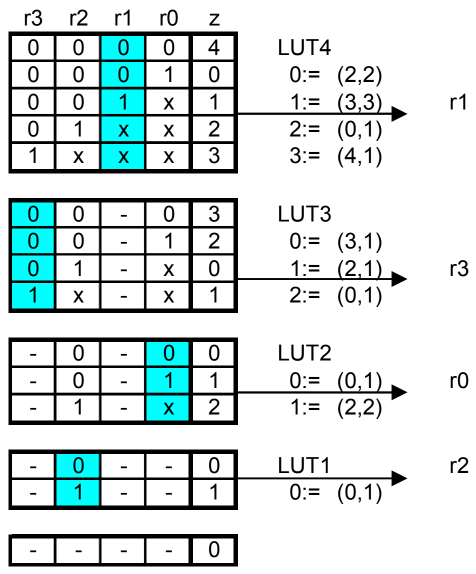
Multiple-output Boolean functions have been more recently represented by BDD_for_CF diagrams [6]. Here the top-down iterative decomposition starts from the root and after a removal of a single variable the whole diagram has to be reconstructed. Another disadvantage of this approach is a large size of BDD_for_CF diagrams.

In this paper we present a heuristic technique of the iterative decomposition of integer-valued functions. Its main contribution is that the bottom-up synthesis of MTBDD/LUT cascade does not require knowledge of optimum ordering of variables, because the order of variables is generated concurrently. Obtained LUT cascades can be used in hardware, firmware and software implementation of combinational and sequential functions.

The paper is structured as follows. Our heuristic approach to construction of sub-optimal MTBDDs and LUT cascades is explained in Section 2. Section 3 deals with three types of arbiters, their decomposition and implementation. Experimental results are summarized in Section 4 and commented on in Conclusion.

2. Construction of LUT cascades and of sub-optimal MTBDDs

In this section we will present a heuristic technique of a sub-optimal LUT cascade construction. It is generalization of the BDD construction by means of iterative disjunctive decomposition [2]. Input variables are selected after one another 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.



r3	r2	r1	r0	z
0	0	0	0	4
0	0	0	1	0
0	0	1	x	1
0	1	x	x	2
1	x	x	x	3

0	0	-	0	3
0	0	-	1	2
0	1	-	x	0
1	x	-	x	1

-	0	-	0	0
-	0	-	1	1
-	1	-	x	2

-	0	-	-	0
-	1	-	-	1

-	-	-	-	0

Fig. 1. Iterative decomposition of an integer function of 4 binary variables (PE4)

Before formulation of the algorithm, we prefer to illustrate the synthesis technique on an example (the 4-input priority encoder PE4). The integer function z = F(r0, r1, r2, r3) of four binary variables is specified by a table with input values from {0,1,x}, Fig.1. We use symbol x to shorten the function specification; regardless whether an associated variable will attain the value 0 or 1, the value at output z will be the same. In the meantime we will select a sequence of input variables for iterative decomposition randomly, e.g. r1, r3, r0, r2. A single variable will be removed from the function in one decomposition step. Starting with variable r1, we inspect column r1 (highlighted at the top table in Fig. 1) to see how many distinct pairs of function values (also equivalently sub-functions of a single variable [2])

$$[F(r0, 0, r2, r3), F(r0, 1, r2, r3)] \qquad (1)$$

are produced by this variable.

We have to analyze pairs of rows in the table such that one row has value 0 in column r1 and another one value 1. Two such rows which are not in conflict (do not have complementary values in the same column except column r1) can be replaced by a single row in the new table of a residual function $F_1(r0, -, r2, r3)$. Bit values 0, 1 and symbol x in columns are combined as shown in Table 1.

Table 1. Combining rows in the function table

row (v = 0)	! v	1	0	x	1	x	0	x	–
row (v = 1)	v	1	0	x	x	1	x	0	–
new row	–	1	0	x	1	1	0	0	–

A symbol “–“ in a certain column means that the variable at that place has already been removed and does not exist. A pair of function values (1) from two rows will be replaced later by a new integer value (id).

There are two rows at the top table in Fig. 1, 1st and 3rd, that can be combined into

00-0 (4,1).

Also the 2nd row can be combined with the 3rd row into

00-1 (0,1).

Two pairs of function values (4,1) and (0,1) with new identities “3” and “2” correspond to two decision nodes at the lowest level of a MTBDD that is just being created, see Fig. 2a. For incomplete functions with don´t care “function value” z = DC we should combine (a, DC) = (DC, a) = (a, a), i.e. replace DC by value a.

If a row contains symbol “x” in column r1, “x” replaced by symbol “–“ will be carried over to the new residual table. Function values (1) are now identical (as above for a z = DC) and variable r1 in fact does not decide anything. A corresponding decision node in the MTBDD has only one output and can be replaced by a shortcut from the input to the output. There are two such rows in the first decomposition step, rows 4th and 5th. They produce degenerate decision nodes 0 and 1 in the lowest level of MTBDD (black dots in Fig. 2a).

By now we have exhausted all possible pairs of rows and have replaced them by new rows in the next (residual) function table. As a result of the removal of variable r1 from the original function, obtained pairs of function values can be assigned the shortest possible code by enumeration. This transformation is shown next to the top table at Fig. 1 and in fact it is realized by the last LUT4 (e.g. for cell input 3, cell output will become 4 or 1 depending on whether r1 is 0 or 1).

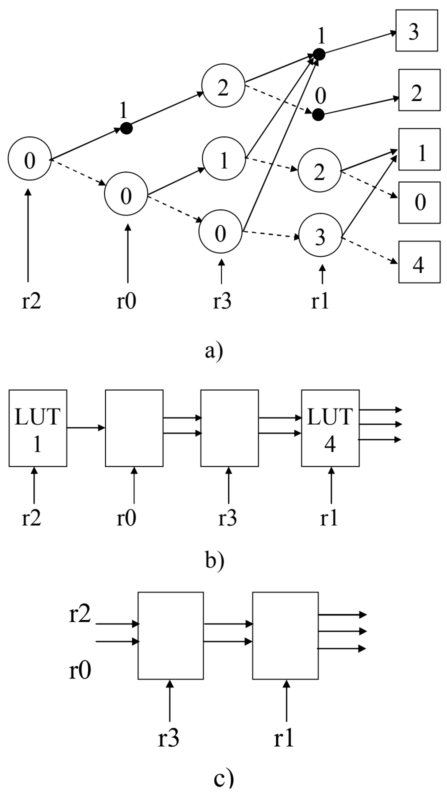


Fig. 2. A MTBDD (a), a generic LUT cascade (b), and a compact LUT cascade with 3-input cells (c) obtained by iterative decomposition (PE4 example)

The same procedure is repeated in the following decomposition steps until all variables will be removed. We proceed in a backward direction, from the leaves to the root of the MTBDD or from LUT4 to LUT1, Fig. 2a, b. In the case of LUT cascades, it is sufficient to go on with iterative decomposition until the number of remaining variables equals to the required number of address inputs to the last LUT found.

The remaining question not addressed as yet is, which variable should be used in any given step. We use a heuristics 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 (including degenerate ones) 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 of true (non-degenerate) decision nodes in the current level of the MTBDD is taken (this corresponds to a minimum number of rows with distinct function values in the pair (1)). In the case of a tie again, a variable is selected randomly.

3. Arbiter circuits

LUT cascades have been applied to many useful digital function modules and their effectiveness and performance has been compared to benchmark circuits [6]. Here we are going to apply LUT cascades to arbiter circuits, i.e. to the area not addressed as yet.

We will synthesize three representative types of arbiters, namely

1. Fixed priority arbiter (also known as a Priority Encoder PE)
2. Round robin arbiter (also known as a Last Granted Lowest Priority scheme, LGLP)
3. Matrix arbiter (Least Recently Served scheme, LRS).

A key property of an arbiter is its fairness. For the purpose of our case study we will understand two concepts of fairness. A weak fairness means that every request is eventually served and with a strong fairness requesters will be served equally often. A traditional design of arbiters is discussed in [7] and [8].

3.1. The priority encoder (PE)

The PE is a combinational circuit that according to a subset of active requests produces the address of the request input with the highest priority in the subset. This is the simplest arbiter of all, but its usefulness in practice is limited because it is not fair, not even in the weak sense. The input request r(n-1) has the highest fixed priority and then the priority decreases to the lowest priority level for input r0. If one request is continuously asserted, none of lower priority requests will ever be served. An n-input PE has $\lceil \log_2 n \rceil + 1$ outputs which provide the address of an active request input with the highest priority. One combination of address bits denotes the case of no active request.

The decomposition of our running example PE4 leads to the MTBDD shown at Fig. 3a and to the cascade of two LUTs. The MTBDD can be simplified to a form at Fig. 3b. This diagram has a very simple linear form and terminal values are generated very early along the main path and do not have to propagate to the end of cascade. Allowing outputs from intermediate cells is a more general version of the binary cascade and can lead to the reduced number of rails between cells in the LUT cascade. Such configuration may be useful e.g. for firmware implementation discussed later on. Comparison of Fig. 2a and Fig. 3a,b shows that the used heuristics really works and reduces cost and width of the MTBDD with a random order of variables in Fig. 2a.

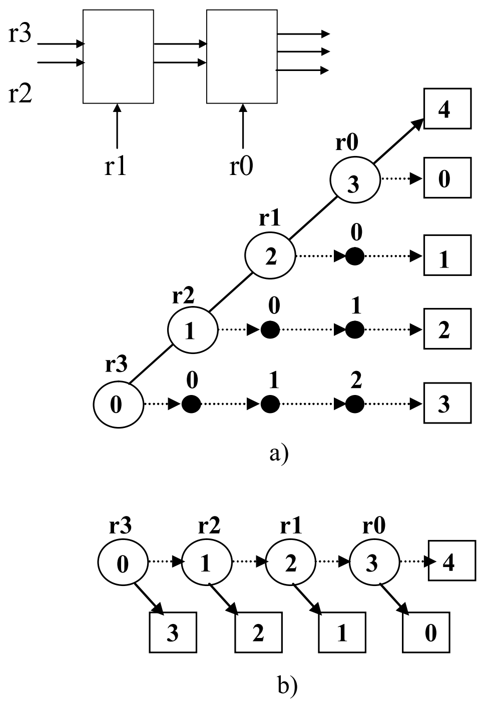


Fig. 3. PE4 arbiter a) MTBDD with degenerate nodes b) MTBDD with terminal values not restricted to the last level

A larger arbiter PE8 with eight request inputs and four address outputs is defined by Table 2. Iterative decomposition leads to the homogenous cascade of a constant width depicted at Fig. 4. Generally, however, the width of a cascade is not constant. At construction of LUT cascades we try to combine adjacent LUTs together to get LUTs of uniform size as often as possible. For example the PE8 cascade at Fig. 4 may be reduced to 3 cells with capacity of 96, 96 and 64 bits.

LUT cascades of this kind can be used for pipelined implementation of general combinational logic systems. The LUT cascade would have to be completed by pipeline registers between cells. These registers would serve also for storing variables used at vertical cell inputs. The performance of the pipeline under the continuous stream of input vectors would then be determined by a single cell delay. One arbitration result would be generated every clock cycle.

Let us note that one way how to implement multi-output LUTs is to compose them out of single-output LUTs. For example LUT cascade at Fig. 5a could be assembled of 16 single-output, 4-input LUTs. Even though HDL synthesis tools can do better than that, the advantage of the cascade is the regular layout that can eventually lead to a smaller occupied area on a chip.

Table 2. The 8-input priority encoder

r7	r6	r5	r4	r3	r2	r1	r0	a3	a2	a1	a0
0	0	0	0	0	0	0	0	1	0	0	0
1	×	×	×	×	×	×	×	0	1	1	1
0	1	×	×	×	×	×	×	0	1	1	0
0	0	1	×	×	×	×	×	0	1	0	1
0	0	0	1	×	×	×	×	0	1	0	0
…	…	…						…	…	…	…
0	0	0	0	0	0	0	1	0	0	0	0

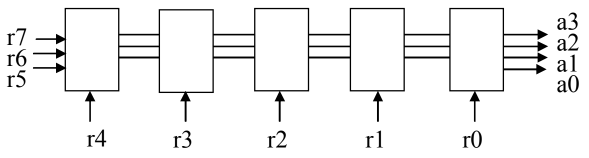


Fig. 4. LUT cascade-based PE8

3.2. The round robin (LGLP) arbiter

This is a Moore type sequential state machine with dynamic priority allocation scheme based on Last Granted Lowest Priority (LGLP) strategy that ensures strong fairness. It has *n* input requests, 2*n* states S0, S1, …, S(2*n*-1) and *n* grant outputs. Even-numbered states monitor request inputs and odd-numbered states generate grant outputs (one grant per state). Priority vector that determines priorities of inputs is modified by cyclic shift in such a way, that the request just satisfied goes to the lowest priority position.

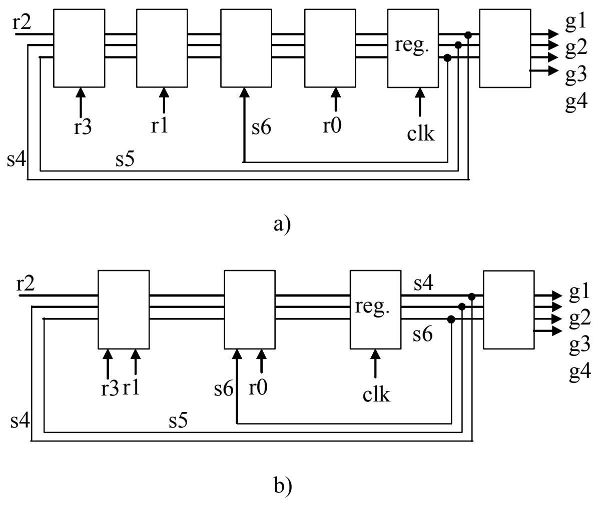


Fig. 5. LUT cascade implementation of the 4-input LGLP arbiter
a) 4-input LUTs b) 5-input LUTs

Let us consider a 4-input LGLP arbiter with 8 states. Priority vectors and grants generated in various states are (the highest priority requests are in bold):

S0: **r3** r2 r1 r0	S1: g3
S2: **r2** r1 r0 r3	S3: g2
S4: **r1** r0 r3 r2	S5: g1
S6: **r0** r3 r2 r1	S7: g0

State transitions for even numbered states are easy to specify; e.g. for state S0 we have

old state	r3 r2 r1 r0	new state
000 (S0)	1 x x x	001 (S1)
000 (S0)	0 1 x x	011 (S3)
000 (S0)	0 0 1 x	101 (S5)
000 (S0)	0 0 0 1	111 (S7)

An odd-numbered state S(2k+1) issuing grant g(3-k) transits to the even-numbered state S[(2k+2) mod 8] as soon as request r(3-k) 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. 5 shows for example two implementations of the LGLP4 that differ in clock speed. Grant signals g1 to g4 are generated from the state code (s6 s5 s4) in the last 3-input cell. Clock speed is determined by the delay of 4 or 2 cells in Fig. 5a and 5b. For comparison, VHDL synthesis tool for Xilinx FPGA generates this arbiter with 17 4-input LUTs in 4 logic levels.

3.3. The matrix arbiter with LRS (Least Recently Serviced) strategy

The LRS strategy cannot be implemented by a dynamically changing priority vector; it has to use priority matrix P, where $p_{ik} = 1$ means that the i-th request has priority over the k-th request [7]. Currently asserted grant output g(j) resets the j-th row 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).

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.

As an example we will use arbiter LRS4 that is implemented in matrix form in [7]. It has 6 state variables s9, s8, s7, s6, s5, s4 and 4 request inputs r3, r2, r1, r0. We will let the LUT cascade generate 4 grant outputs g3, g2, g1, g0, which will be then used to reset and set selectively 6 state flip-flops.

This time we have implemented LRS4 arbiter in firmware. We have used HIDET tool (described below) to decompose function LRS4: $(Z_2)^{10} \rightarrow Z_4$. The result is shown as MTBDD at Fig. 6.

Evaluation of Boolean functions at the firmware level can use nicely the LUT cascade paradigm. By making use of hardware micro-engines 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 [2], is depicted in Fig. 7.

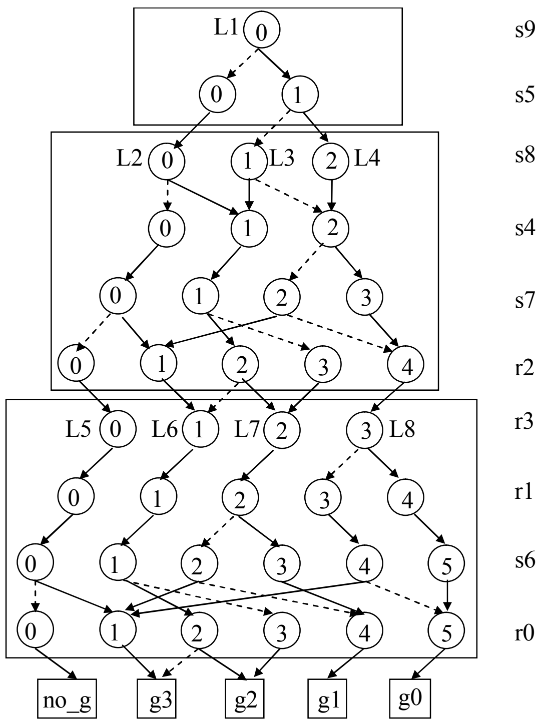


Fig. 6. MTBDD of the 4-input LRS arbiter

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

1) the jump to an address specified in micro-instruction and modified by BCU; symbolic μI :
   Ln: exit Lm@x1...xk;
2) conditional output and the jump to an address specified in micro-instruction (no modification),
   Ln: c_output exit Lm.

The first format includes jumps to the target address obtained from the address specified in the micro-instruction; this latter address is modified by external variables, by up to 4 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 field.

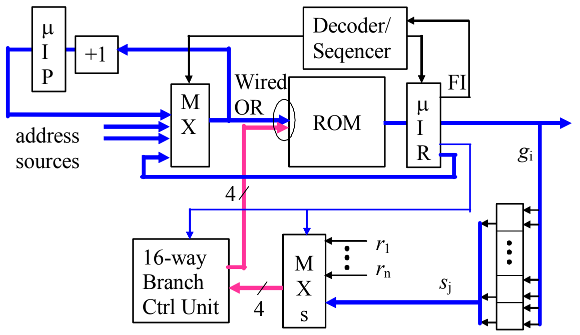


Fig. 7. Micro-programmed controller architecture with multi-way branching

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. This vector is then wire-ORed with the address obtained from the micro-instruction. 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 first reset to 0.

```
LRS: exit L1@s9s5
L1@00: exit L2@s8s4s7r2
L1@01: exit L2@s8s4s7r2
L1@10: exit L3@s8s4s7r2
L1@11: exit L4@s8s4s7r2
L2@0000: exit L5@r3r1s6r0
L2@0001: exit L5@r3r1s6r0
L2@0010: exit L6@r3r1s6r0
…..
…..
L8@1111: g0 exit Next
L8@1110: g0 exit Next
Next:
```

Fig. 8. A symbolic microprogram for the LRS4 arbiter

If there are more than 4 external variables, LUT cascade paradigm is used. We will illustrate rewriting a MTBDD at Fig. 6 into the micro-program with multi-way branching. The symbolic micro-program targeted for the micro-engine in Fig. 7 is shown in Fig. 8. The micro-program is composed of 8 dispatch tables, one of size 4 (node L1) and 7 of size 16. The total number of micro-instructions is thus $4 + 7 \times 16 + 1 = 117$ 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:

$R4 = R5 = R6 = g0$, $R7 = R8 = g1$, $R9 = g2$,
$S4 = g1$, $S5 = S7 = g2$, $S6 = S8 = S9 = g3$.

Out of 6 state variables and 4 input requests up to 4 signals are selected by 4 multiplexers, fed into BCU and used in the least significant positions for address modification, as shown in Fig. 7.

Had we used only single variable tests (a binary program with 2-way branching), we would need 40 dispatch tables of size 2, i.e. 80 microinstructions in total. However, the performance would be almost 3 times lower due to execution of 11 microinstructions, one in each level of the MTBDD, for one decision.

4. Experimental results

To aid LUT cascade synthesis, the program tool HIDET (Heuristic Iterative DEcomposition Tool) has been developed. It basically implements the following algorithm (letters S stand for sets, M and L for tables, w for local MTBDD width and d for "discount"):

```
Input:
    $M_{in}$, input function table;
    $S_v$, the set if input variables
    $n = |S_v|$, number of input variables;
Output: i in 1 to n
    $M_i$, function tables ;
    $L_i$, LUTs;
    $v_i$, variable removed in step i ;

Initialize $i \leftarrow 1$, $M_0 \leftarrow M_{in}$;
for i in 1 to n do
    // Determine the best variable
    $v_{best} \leftarrow$ arbitrary variable from $S_v$,
    $w_{best} \leftarrow$ size($M_{i-1}$), $d_{best} \leftarrow 0$;
    for all variables $v \in S_v$ do
        $M_p \leftarrow$ make_pairs($M_{i-1}$, v);
        $S_p \leftarrow$ unique_pairs($M_p$);
        $S_m \leftarrow$ merge_compatible_pairs($S_p$);
        $w \leftarrow$ size($S_m$);
        $d \leftarrow$ number of constant pairs in $S_m$;
        if ($w < w_{best}$) or ($w == w_{best}$ and $d > d_{best}$) then
            $v_{best} \leftarrow v$, $w_{best} \leftarrow w$, $d_{best} \leftarrow d$;
        endif
    endfor
    $v_i \leftarrow v_{best}$;
    // Decompose
    $M_p$ ← make_pairs($M_{i-1}$, $v_i$);
    $S_p$ ← unique_pairs($M_p$);
    $S_m$ ← merge_compatible_pairs($S_p$);
    $L_i$ ← enumerate_pairs($S_m$);
    $M_i$ ← replace pairs in $M_p$ by new id numbers in $L_i$;
    $S_v$ ← $S_v \setminus \{v_i\}$;
endfor
```

Sequential as well as parallel (OpenMP) versions of the program were compiled, ran and gave the same results. The parallel version was tested on 4- and 8-cores SMPs with a speedup about 80%. We could not test the program on a standard benchmark set, because most of the benchmark circuits used to be specified by function tables with overlapping sub-domains (rows). As yet, HIDET tool can process only disjunctive sub-domains; the next version of HIDET should address the more general case, too.

Function tables of many instances of three types of arbiters have been generated and then processed by HIDET tool. Iterative decomposition of this class of functions was a matter of seconds (LRS10). The sample results for LGLP and LRS arbiters are summarized in Table 3. PE arbiters are the easiest to decompose and have not been included; optimum ordering of variables for PEs copies the one from the headings of the function table, Fig. 4. Parameters of generic cascades are displayed, cell after cell, from left to right (i.e. in the opposite order than they were obtained by the decomposition procedure) in Table 3.

Each column represents one cell, from the top down:

1. index of a variable (higher indices: state variables, lower indices: request inputs)
2. the number of all rows in the LUT (the local width of the MTBDD)
3. the number of LUT rows with the same values in pair (1) (degenerate decision nodes).

Table 3 thus describes profiles of generic cascades.

From Table 3 one can obtain two global parameters of the cascades: cost (the number of true decision nodes) and the maximum number of rails between cells, Table 4. A large variety of LUT cascades can be designed, examples given in column "LUTs" use the same size LUTs with specified cascade length x number of inputs (in). It is interesting to compare these results with designs obtained by Xilinx FPGA synthesis tool. The number of 4-input LUTs compares to the number of true decision nodes (cost); one decision node can be mapped to a 3-input LUT and can be taken approximately as one half of the 4-input LUT. Also, 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. This way it comes out that we should be able to get the same or better performance with LUT cascades.

Table 3. Parameters of generic LUT cascades for LGLP and LRS arbiters

Legend:
← index of variable
← # all LUT rows
← # LUT rows [x,x]

$\boxed{0}$ = last LUT
13 → 4 rails in
var 0 removed first

LGLP3

4	3	1	2	5	0
1	2	4	6	8	9
0	0	1	2	3	4

LGLP4

5	4	2	3	1	6	0
1	2	4	6	9	13	13
0	0	0	2	5	2	6

LGLP6

7	6	0	1	9	5	2	3	4	8
1	2	4	6	9	13	15	16	18	17
0	0	1	2	0	7	7	7	9	8

LGLP8

10	8	7	5	6	9	4	3	2	1	0	11
1	2	4	6	8	11	14	16	19	22	25	25
0	0	1	3	3	3	9	10	12	14	15	8

LGLP10

11	10	9	8	6	7	12	5	4	3	13	2	1	0	14
1	2	4	6	8	11	15	19	23	27	31	32	32	35	25
0	0	1	3	4	6	2	13	17	19	6	16	15	17	17

LRS3

5	2	4	1	3	0
1	2	3	3	4	5
0	1	1	2	1	4

LRS4

9	5	8	4	7	2	3	1	6	0
1	2	3	3	4	5	4	5	6	6
0	1	1	2	1	4	3	4	2	5

LRS6

20	14	19	13	18	9	12	8	17	5	11	7	16	4	2	10	6	3	15	1	0
1	2	3	3	4	5	4	5	6	6	5	6	7	8	7	6	7	8	9	9	8
0	1	1	2	1	4	3	4	2	5	4	5	2	7	6	5	6	7	4	8	7

The area for wiring LUT cascades promises to be much lower, whereas the area of cascaded LUTs themselves slightly higher due to their coarser granularity. The large area for the interconnections in an FPGA is thus absorbed in the larger LUTs in the cascade.

5. Conclusions

The presented method of LUT cascade synthesis of multiple-output Boolean functions aided by HIDET tool proved to be suitable for synthesis of combinational and sequential designs with tens of input/output and state variables. Arbiters, as well as other digital systems frequently used in practice, have relatively low complexity, what 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 such as SECDED are also applicable. 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.

Table 4. Comparison of FPGA designs and LUT cascades

	FPGA		LUT cascade		
	#4-LUT	levels	cost	width	LUTs
LGLP3	10	3	20	≤4	2 x 5 in
LGLP4	17	4	33	≤4	3 x 5 in
LGLP6	48	6	60	≤5	4 x 6 in
LGLP8	70	9	75	≤5	6 x 6 in
LGLP10	122	8	135	≤6	6 x 7 in
LGLP16	433	17	194	≤6	5 x 9 in
LRS3	6	2	10	≤3	2 x 4 in
LRS4	8	2	17	≤3	2 x 6 in
LRS6	24	3	33	≤4	3 x 9 in

Effectiveness of LUT cascades can be derived from the size of the complete function table and aggregate capacity of all LUTs in a cascade. E.g. LGLP8 arbiter with 12 input variables requires 2^{12} x 4 bits, whereas the LUT cascade exemplified in Tab. 4 only less than 6 x 2^7 x 5 bits (23%). Most of the functions that occur in digital design are decomposable effectively into cascades. One exception is the class of binary multipliers: for all possible variable orderings is the BDD size exponential for n-bit inputs and 2n-bit output [9].

Future research should address a more general specification of multiple-output Boolean functions, namely in a form of Boolean expressions representing individual binary outputs. Also the quality of heuristic optimization of variable ordering should be put under a test against results obtained by exhaustive testing of all permutations of variables. The effectiveness of LUT cascades could be improved further by creating smaller groups of binary outputs and decomposing separately each group. Appropriate design techniques could provide cost-effective cascades for new classes of functions. Security and safety oriented applications will be the nearest target.

6. References

[1] K. Nakamura, T. Sasao, M. Matsuura, K. Tanaka, K. Yoshizumi, H. Qin, and Y. Iguchi: 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*, April 20-22, Yokohama, Japan, 2005.

[2] V. Dvořák: LUT Cascade-Based Architectures for High Productivity Embedded Systems, In: *International Review on Computers and Software*, Vol. 2, No 4, Naples, Italy, pp. 357-365, 2007.

[3] M. Yoeli: The Synthesis of Multivalued Cellular Cascades. *IEEE Trans. On Computers*, Vol. C-9, pp. 1089-1090, Nov. 1970

[4] V. Dvořák: An optimization technique for ordered (binary) decision diagrams, *Proceedings of the 6th Annual European Computer Conference CompEuro' 92*, Hague, NL, pp. 1-4, 1992

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

[6] T. Sasao and M. Matsuura: BDD representation for incompletely specified multiple-output logic functions and its applications to functional decomposition, *Design Automation Conference*, pp.373-378, June 2005.

[7] W.J. Dally, B.Towles: *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers / Elsevier, San Francisco, CA, 2003.

[8] E. S. Shin: Automated Generation of Round-Robin Arbitration and Crossbar Switch Logic. *PhD Thesis*, School of Electrical and Computer Engineering, Georgia Institute of Technology, November 2003.

[9] R.E.Bryant: On the complexity of VLSI implementations and graph representations of Boolean functions with applications to integer multiplication. *IEEE Transactions on Computers*, Vol. 40, pp.205–213, 1991.

Acknowledgement

This research has been carried out under the financial support of the research grants “Design and hardware implementation of a patent-invention machine”, GA102/07/0850 (2007-9), “Safety and security of networked embedded system applications”, GA102/08/1429 (2008-10), both care of Grant Agency of Czech Republic, and “Security-Oriented Research in Information Technology”, MSM 0021630528 (2007-13).