Efficient Evaluation of Multiple-Output Boolean Functions in Embedded Software or Firmware

Vaclav Dvorak
Brno University of Technology, Brno, Czech Republic
Email: dvorak@fit.vutbr.cz

***Abstract* — The paper addresses software and firmware implementation of multiple-output Boolean functions based on cascades of Look-Up Tables (LUTs). A LUT cascade is described as a means of compact representation of a large class of sparse Boolean functions, evaluation of which then reduces to multiple indirect memory accesses. The method is compared to a technique of direct PLA emulation and is illustrated on examples. A specialized micro-engine is proposed for even faster evaluation than is possible with universal microprocessors. The presented method is flexible in making trade-offs between performance and memory footprint and may be useful for embedded applications where the processing speed is not critical. Evaluation may run on various CPUs and DSP cores or slightly faster on FPGA-based micro-programmed controllers.**

***Index Terms* — Embedded software, Boolean function evaluation, Binary Decision Diagrams, LUT cascades**

I. Introduction

Efficient evaluation of Boolean functions is an important part of many embedded software systems. Functions most frequently used in embedded system practice are not random, but application-specific with low complexity. Among them sparse functions defined below include applications such as encryption, data compression and conversion, pattern matching and searching, moving window functions on data streams, etc. We will address Boolean functions of large numbers (tens, hundreds) of variables because small size systems can be implemented directly in hardware, e.g. in PLA, ROM or TCAM (Ternary Content Addressable Memory).

Software implementation of Boolean functions will be assumed in a form of a data structure describing the function and of a compiled program that reads the input vector and evaluates the function with the use of this data structure. The size of the code and of the data structure is one figure of merit and the other is the evaluation time from reading the input to generating the output.

Hereafter we will use three compact representations: a PLA-like table, Look-Up Tables (LUTs) and binary decision diagrams (BDDs). The BDDs are well known, especially the reduced ordered BDDs (ROBDDs), [1]. On the base of ROBDDs we will develop a more practical representation – cascades of LUTs.

Software implementation of Boolean functions has been up to now studied especially in connection with PLCs ("ladder diagrams") [2], digital system simulation, formal verification and testing [1], or specialized event processing [3], where either a speed (PLC) or a required memory were not that important. On the contrary, in embedded systems we do care for performance, memory space as well as for power consumption. We will demonstrate that presently used algorithms (PLA emulation, a BDD traversal or evaluation of Boolean expressions) are generally too slow and that the use of LUT cascades enables faster evaluation. The longer cascades with simpler LUTs are slower than shorter cascades with larger LUTs, and thus the processing speed can be even adjusted to requirements.

The paper is structured as follows. In the following Section II we introduce terminology and notation concerning Boolean functions; a traditional approach to Boolean function evaluation is assessed in Section III. Binary decision diagrams (BDDs) and LUT cascades are introduced in Section IV, together with some complexity issues. Variable ordering in MTBDDs and in LUT cascades is a subject of Section V. In Section VI we give examples of LUT cascades for sample Boolean functions and illustrate trade-offs between speed of evaluation and required memory space. A micro-engine for LUT cascade processing is presented in Section VII. Results obtained with selected functions and some generalizations are commented on in Conclusions.

II. Terminology and Notation

To begin our discussion, we define the following terminology. A system of m Boolean functions of n Boolean variables,

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

will be simply referred to as multiple-output Boolean function F_n with output values from $Z_R = \{0, 1, 2, \ldots, R\text{-}1\}$,

$$F_n: (Z_2)^n \rightarrow Z_R\,, \tag{2}$$

where R is the number of distinct combinations of m output binary values enumerated by values from Z_R.

Function F_n is incomplete if it is defined only on set $X \subset (Z_2)^n$; $(Z_2)^n \setminus X = D$ is the don't care set.

The behavior of a combinational circuit can be described by the system of m explicit complete functions

Based on "Time- and Space-Efficient Evaluation of Sparse Boolean Functions in Embedded Software", by Vaclav Dvorak, which appeared in the Proceedings of the 14th IEEE Int. Conf. and Workshops on the Engineering of Computer-Based Systems 2007, Tuscon, AZ, USA, March 2007. © 2007 IEEE.

of n variables

$$y_i = f_n^{(i)}(x_1, x_2, \ldots, x_n), \qquad i = 1, 2, \ldots, m \qquad (3)$$

or $\boldsymbol{y} = \boldsymbol{F}(\boldsymbol{x})$ in vector notation. Alternative implicit description is based on the so called output characteristic function [7]

$$\phi_0(\boldsymbol{x}, \boldsymbol{y}) = 1. \qquad (4)$$

Machine representation of Boolean functions uses binary decision diagrams (BDDs), which can have many forms. Bit-level binary decision diagrams (BDDs), ordered binary decision diagrams (OBDDs) and reduced ordered binary decision diagrams (ROBDDs) are well known representation of a single Boolean function in a form of a directed acyclic graph [1]. Whereas ROBDD is a canonical (unique) representation for any given complete function and an order of variables, incomplete Boolean functions may be transformed into more than one complete form and into the associated ROBDD.

Important parameters of a BDD are its size and width, i.e. the total number of decision nodes and the maximum number of edges between adjacent levels, where the edges pointing to the same nodes are counted as one. The size determines the memory space needed to store the BDD data structure while the width K (also C-measure, [4]) determines a BDD form factor since the height is given by the number of variables. The construction of minimum-size or by the same token minimum-width ROBDDs belong among NP-complete problems [5]; the size and width of the ROBDD depend on variable ordering and there are n! possible orderings of n variables. A heuristic approach can be used in a search for near-optimal orderings [6]. Upper bounds on the OBDD's size and width for general random complete Boolean functions grow exponentially with number of variables n for any ordering, but functions used in digital systems design with few exceptions do have a reasonable BDD size and small width.

M-ary decision diagrams are straightforward generalization of BDDs. They have two types of nodes: decision and terminal nodes. Decision node L is testing M-ary variable var(L) and its outgoing edges are marked by its values 0, 1, …, M-1. The terminal node assigns a single value from Z_M (generally Z_R, $R \neq M$) to output $y = F_n(x_1, x_2, \ldots, x_n)$.

To represent a system of Boolean functions (1) by means of decision diagrams, we can use either m bit-level BDDs, one for each of m Boolean functions (possibly sharing some of their sub-diagrams in Shared BDDs or SBDDs, [7]), or one word-level BDD (WLBDD) with n Boolean decision variables and with R integer terminal values. There are many types of WLDDs. Multi-terminal BDDs have integer leaves and therefore represent functions from Booleans to integers. A BMD (Binary Moment Diagram) is more compact representation for some useful arithmetic functions which have exponential size if represented by MTBDDs. Hybrid decision diagrams HDDs are a combination of MTBDDs and BMDs.

BDD for Characteristic Function (BDD for CF) [7] is yet another representation of multiple-output functions, which uses the shortest encoding of output vectors $\boldsymbol{y}$ by means of auxiliary variables. The advantage of the BDDs for CF is that tools useful for optimization of BDD for a single Boolean function (4) can be used without modification for multiple-output functions as well.

As the LUT cascades are the main concern of this paper, we will provide a formal definition. A LUT will be also interchangeably referred to as a "cell".

Def. 1. A cascade C of a form $k \times m$ is the system

$C = [\, K, M, H_1, H_2, \ldots, H_B, \mu]$

where

$K \leq 2^k$ ($M \leq 2^m$) is the number of specified Boolean input vectors at k horizontal (m vertical) cell inputs,

H_i: $(Z_2)^k \times (Z_2)^m \rightarrow (Z_2)^k$, $1 \leq i \leq B$ are functions implemented by individual cells,

B, the cascade length, is the total number of cells and

μ: $\{1,2,\ldots, \mathrm{B}\} \rightarrow (Z_2)^m$ assigns m-tuples of input variables x_i, $i = 1,2,\ldots, n$ to B cells in the cascade.

The above cascade has the width k horizontal rails carrying Boolean values and each cell has m vertical (side) inputs. The last cell in the cascade may have $r \neq k$ outputs.

Def.2. A cascade is said to be non-redundant if each variable used at 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. The attribute "redundant" will be used always explicitly.

Note. Cascades considered in [16] use cells with additional vertical outputs. These intermediate outputs can reduce the cascade width k. Intermediate outputs are either individual Boolean variables y_i as in (3) or the complete integer values from Z_R as in (2); in the latter case BDD leaves appear not only in the bottom of the diagram, but span more its levels. A consequence for software evaluation is that some partial Boolean outputs or function values are generated earlier than others.

III. Traditional approaches to evaluation of Boolean functions: PLA emulation

Hardware implementation of Boolean functions in Programmable Logic Array (PLA) can serve as an initial prototype for software implementation. PLA consists of AND-matrix and OR-matrix. Rows of the AND-matrix define terms and OR-matrix serves for accumulation some of them into the binary outputs, Fig.1.

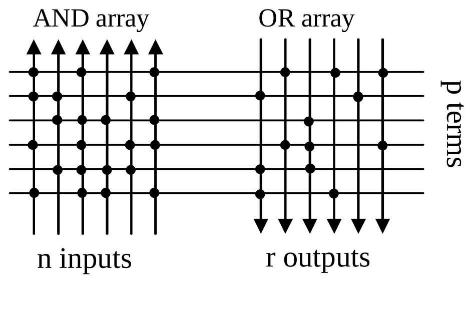


Fig.1. Structure of PLA

The set of p terms produced by AND-matrix (a term vector) can be generated in parallel, each term in one bit of the computer word. If the capacity w bits in a single word is not enough, $\lceil p/w \rceil$ computer words can be used to accommodate all the terms. The terms are evaluated in n steps, one input Boolean variable at a time. Two masks $m0(x)$ and $m1(x)$ of length p bits are maintained for each variable x. The masking bit of mask $mv(x)$, $v = 0,1$, in a position of term t is denoted $mv(x, t)$ and has the following value:

if x occurs in t, then $mv(x, t) = v$
if $!x$ occurs in t, then $mv(x, t) = !v$
if x does not occur in t, $mv(x, t) = 1$.

Two masks for each variable are generated only once, at the beginning, based on their occurrence in PLA terms. The term vector is initialized to all ones and then a sequence of masks is applied to it using the bitwise logical AND operation. For variable x mask $mv(x)$ is used depending on the input value $x = v$. All the terms are thus updated in parallel by the bitwise AND operation and the (full) width of computer word is utilized.

As soon as all terms are ready, we have to emulate OR-matrix – apply bitwise OR operation selectively to certain bits. Another set of r masks will be used for r outputs. Unused terms in the term vector are masked out and if at least a single 1 remains, the result TRUE. The memory size for storing all sets of masks is thus

$$\text{space} = (2n + r)\lceil p/w \rceil \text{ words} \qquad (5)$$

and time complexity is

$$\text{time} = C_1 n + C_2 r, \qquad (6)$$

where C_1 and C_2 are execution times in clock cycles related to mask applications.

If the number of terms p is less than the number of variables n, a dual evaluation method may be more advantageous. The relevant terms are generated one after another from the input vector using again two sets of masks. As soon as the term vector is assembled, the outputs are generated the same way as before.

Similarly, if $p < r$, we can create the output vector faster by ORing p partial vectors $mv(t)$; partial vector $mv(t)$, $v = 0,1$, is selected if term $t = v$. Space and time complexities under all conditions are:

$$space = \begin{Bmatrix} 2n\lceil p/w \rceil \\ 2p\lceil n/w \rceil \end{Bmatrix} + \begin{Bmatrix} r\lceil p/w \rceil \\ 2p\lceil r/w \rceil \end{Bmatrix} \qquad (7)$$

$$time = \begin{Bmatrix} C_1 n \\ C_3 p \end{Bmatrix} + \begin{Bmatrix} C_2 r \\ C_4 p \end{Bmatrix} \qquad (8)$$

$$conditions: \begin{Bmatrix} p \geq n \\ p < n \end{Bmatrix} + \begin{Bmatrix} p \geq r \\ p < r \end{Bmatrix}$$

Example:

Size of full tables for two sample PLAs is given:

TABLE 1.
PARAMETERS OF PLA1 AND PLA2

	n	r	p	|X|	size [B]
PLA1	13	8	31	175	8192
PLA2	11	8	53	632	2048

Data structures for emulation of two PLAs with w = 8 bits according to (5) take up only 156 and 268 bytes, respectively. Time complexity is $n + r = 21$ and 19 time steps provided that $C_1 \approx C_2$. (End of example.)

IV. LUT CASCADES, BDDS AND COMPLEXITY ISSUES

A. *Relation of MTBDDs and LUT Cascades*

Whereas BDDs and MTBDDs proved useful in many areas of digital design [8] where they provide compact data structures and a degree of flexibility in manipulating them, they are not as wonderful for the purpose of function evaluation. The primary reason is the slow speed, since the evaluation process inspects one Boolean variable after another. There is though a certain speedup in comparison to direct evaluation of Boolean expressions, because each variable is processed only once. Straightforward remedy how to speed up the traversal of a BDD is to process several variables at a time. This way we will derive LUT cascades, in fact a special case of LUT networks.

A close relation between both these representations of multiple-output Boolean functions will be illustrated on a bit-counting example. The function F_n: $Z_2^n \rightarrow Z_n$ counts the number of 1´s presented at n inputs and represents it by a binary number. The MTBDD and associated LUT cascade are displayed in Fig. 2 for $n = 4$. Generalization for larger values of n is easy. As the number of nodes grows linearly from the root to leaves, the width of the MTBDD is given by the last level of decision nodes and has the value of $K = n$.

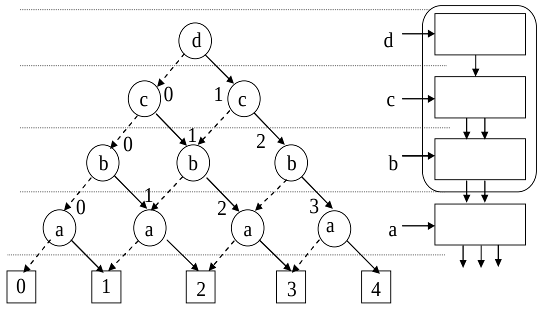


Fig.2. Bit counting example

What connects two representations is the concept of sub-functions. Informally, the sub-function f of F_n is a function of s variables obtained from F_n by setting $n - s$ variables to fixed constant values. The number of distinct sub-functions of s variables, $s = 1, 2,\dots,$ n-1, the so called profile, characterizes complexity of the Boolean function. In Fig.2 we can recognize distinct sub-functions as edges crossing boundaries between MTBDD layers, counting edges incident with the same node only once. Edges are labeled by ID codes of distinct sub-functions. From the top down, there are 2 sub-functions of variables a, b, c (ID codes 0, 1), 3 sub-functions of variables a, b (ID codes 0, 1, 2), 4 sub-functions of variable a (ID codes 0, 1, 2, 3), and 5 sub-functions of zero variables (constant terminal values 0 to 4). LUT contents are defined by binary ID codes and a side variable entering a cell, and binary ID codes generated by the cell. Co-synthesis of MTBDD and LUT cascade will be presented in Section V.

As can be seen, the difference between the MTBDD and the LUT cascade is in communication among the MTBDD layers and LUTs in the cascade: in MTBDD each sub-function ID code requires an individual edge ("wire"), whereas the ID codes being sent between LUTs are binary coded. The number of rails k in the cascade is therefore

$$k = \log_2 \lceil K \rceil. \tag{9}$$

This difference of two representations reflects itself in the way how the program interprets a certain application-specific MTBDD or a LUT cascade. In case of the MTBDD we may use for each node a record with 3 fields. A format indicator is one-bit field specifying the leaf node (leaf nodes may generally occur at any level of the diagram). Two other fields of the leaf node are then used for output. If the node is not a leaf, two fields (adjacent words) contain pointers to the base addresses of other nodes. The base address is then modified by the value of a current control variable(s) and is used to extract the correct field with the pointer to the next node. The program traverses a certain path in the MTBDD from the root to a leaf in at most n steps.

LUTs are interpreted similarly, only the pointer to the next LUT is obtained from the current output by concatenating it with the control variable value and adding it up to the next LUT base address. If suitable, some LUTs can be combined to provide even faster processing (see first three cells in Fig.2 combined into one).

B. Complexity isues

Many questions arise in connection with implementation of the given multiple-output Boolean functions by LUT cascades. In our bit-counting example the number of sub-functions from the root to leaves (a profile) was increasing linearly. Some other functions may have a profile almost constant, what makes the number of rails in the cascade also constant – a desirable feature. However, for randomly generated functions and for multipliers the profile and maximum BDD width K increases exponentially (parameter $k = \log_2 \lceil K \rceil$ linearly) with the number of variables. The question is, what will be the required number of cascade rails in general case. If we remove the restriction that each input variable can be used only once, the result is available as Theorem 1 for multi-valued redundant cascades (repeated BDDs):

Theorem 1. [9] Every function

$$F_n: (Z_M)^n \to Z_R, \qquad M \geq 2,\ R > 2$$

is realizable by the LUT cascade with B_n cells (with K-valued signals between adjacent cells and external M-valued signals on side inputs).

If R = 3, 4, 5, 7, 8, 9, 11, … then $K = R$ else if R = 2, 6, 10,…, 2(2t+1), …, then $K = R+1$; (t=0,1,2,..). $B_n = \mathrm{M}(B_{n-1}+1)$, $B_1=1$.

Synthesis method based on Theorem 1 is not practical, as it produces too long cascades and of the same length for functions with different complexity. However, redundant use of variables may sometimes be useful to reduce the number of cascade rails. Some examples are given in [11].

The size of the ROBDD for a single Boolean function of n variables given by Boolean expression in DNF is known to be less than the number of literals in the expression. More accurately is the BDD size upper-bounded by [10]

$$P \leq \min_k \left\{ L - \left\lceil k \tfrac{L}{n} \right\rceil + 2^k - 1 \right\},\quad k = 1,2,...,L, \tag{10}$$

where L is the number of literals in DNF.

Complexity of MTBDDs for general multiple-output Boolean functions gives the following

Theorem 2 [11]:

Size P and width K of the MTBDD for function F_n: $(Z_2)^n \to Z_R$ are upper-bounded by

$$\begin{aligned} P &\leq \min_i (2^{n-i} + R^{2^i}) - R - 1 \\ K &\leq \max_i \left[\min(R^{2^i}, 2^{n-i}) \right] \end{aligned} \tag{11}$$

The width K of MTBDD and thus the cascade width k (a number of rails) have reasonably low values for many functions arising in practice. One class of such functions is defined below.

Def. 3. Sparse functions.

Under the *sparse functions* F_n: $(Z_2)^n \to Z_R$ we will understand functions with the domain $(Z_2)^n$ divided into two parts X and D, $(Z_2)^n = X \cup D$, $|X| << 2^n$, if one of the following conditions hold:

1) F_n is an incomplete function in $(Z_2)^n$, F_n: $X \to Z_R$ and $(Z_2)^n \setminus X = D$ is the don't care set.
2) F_n: $X \cup D \to Z_R$. Mapping $D \to Z_R$ is artificially defined to make implementation as easy as possible.
3) F_n is a fully specified function in $(Z_2)^n$,
   F_n: $[X \to Z_R \setminus \{0\}, D \to \{0\}]$

(without loss of generality, value 0 is taken as the dominant value).

Functions in three above classes are quite common in digital design.. They have low BDD width and are suitable for LUT cascade implementation. ROBDDs can be obtained by applet [12] and LUT cascades can be quickly derived by slicing ROBDDs. Complexity of LUT cascades for sparse functions is established by the following theorem [11].

Theorem 3.

Every R-valued incomplete function of n binary variables

$$F_n: X \to Z_R,\ X \subset (Z_2)^n$$

is realizable as the output function of the LUT cascade with $k \leq \lceil \log_2 |X| \rceil$ rails.

Every R-valued complete function of n binary variables

$$F_n: X \cup D \to Z_R \quad [X \to Z_R \setminus \{0\}, D \to \{0\}]$$

is realizable as the output function of LUT cascade with $k \leq \lceil \log_2 (|X| + 1) \rceil$ rails.

Another class of sparse functions is defined below.

Def.4. Multiple-output Boolean functions are symmetric if they are invariant under any permutation of inputs. Their values depend only on the number of active inputs.

Theorem 4. MTBDD width K of symmetric functions is $K = n$, the number of Boolean variables.

Proof: It is clear, that all sub-functions of a symmetric function are also symmetric. There are only up to $n-s+1$ distinct sub-functions of s free variables corresponding to $n-s$ fixed variables at all 0´s, single 1, two 1´s, three 1´s, …, all 1´s. Since $1 \leq s \leq n-1$, maximum MTBDD width $K = n$ is at the lowest level where $s = 1$.

C. *Example of a sparse function*

Boolean function of $n = N \times N$ variables returns 1 for every configuration of 1's (queens) on the $N \times N$ chessboard, such that no queen attacks another one. E.g. for $N = 4$ two solutions at Fig.3 can be generated by Boolean function F_{16}

$$F_{16} = !a_{11}*!a_{12}*a_{13}*!a_{14}*a_{21}*!a_{22}*!a_{23}*!a_{24}*!a_{31}*!a_{32}* !a_{33}*a_{34}*!a_{41}*a_{42}*!a_{43}*!a_{44} + !a_{11}*a_{12}*!a_{13}*!a_{14}*!a_{21}* !a_{22}*!a_{23}*a_{24}*a_{31}*!a_{32}*!a_{33}*!a_{34}*!a_{41}*!a_{42}*a_{43}*!a_{44} \qquad (12)$$

a11	a12	a13	a14
a21	a22	a23	a24
a31	a32	a33	a34
a41	a42	a43	a44

Fig.3. Two solutions of 4-queens problem

Function F_{16} is a sparse function (sub-class 3); number of literals in Boolean expression is $L = 32$, $n = 16$, the size is according to (10) upper bounded by 31 nodes and the width is upper-bounded by $\lceil \log_2 3 \rceil = 2$ according to Theorem 3. The real ROBDD generated by the applet [12] has 29 nodes and is shown in Fig.4. The notation used in (12) corresponds to applet [12] and to Fig.3, but variables in the ROBDD at Fig. 4 are enumerated from 0 to 15.

By slicing the ROBDD horizontally as shown in Fig.4, we obtain the LUT cascade. Two sub-function ID codes plus constant 0 are transferred between BDD layers, so that 2-bit code will do. Possible configurations of LUT cascades are in Fig.5.

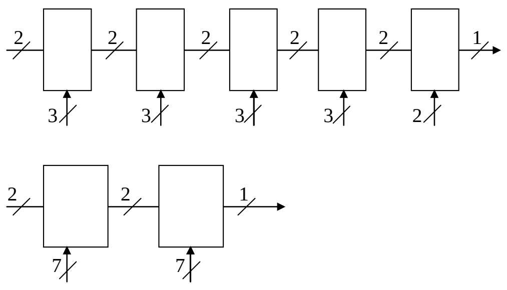


Fig. 5. LUT cascades for 4-queens problem (© IEEE 2007)

The most efficient representation and evaluation of F_{16} is thus clear: two memory words are sufficient to store two min-terms in (12) and two bitwise comparisons will do for the quickest evaluation. In case of LUT cascades, two table look-ups are sufficient in the shorter LUT cascade in Fig. 5, giving a similar speed as with two bitwise comparisons. However, memory consumption is worse, $512 \times (2+1)$ bits against 2 words, 16 bits each.

Now if we move to 8-queens problem, there will be 92 solutions described by F_{64}. Solutions can be found by the known algorithm [13]. To store 92 binary vectors of length 64 is still acceptable, but instead of a linear search we can order solutions and do better with the logarithmic search in $\lceil \log_2 92 \rceil = 7$ steps at most.

The ROBDD size is according to (10) upper-bounded by 5535 nodes, so that storing of such BDD would not be space efficient at all. A traversal of this ROBDD would need 64 steps in the worst case, what is bad as well. A LUT cascade could be faster, but at the cost of total memory capacity for LUTs.

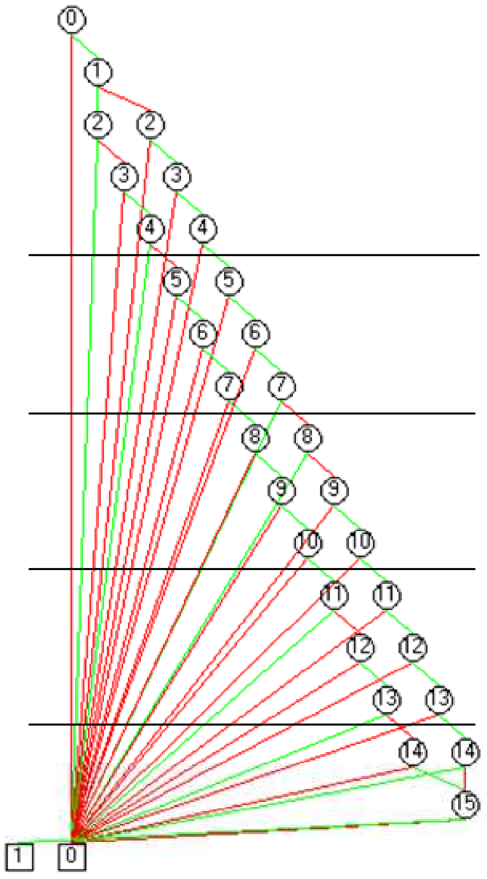


Fig.4. ROBDD for 4 queens problem (© IEEE 2007)

V. Variable ordering for minimum width (size) of the MTBDD

Design of LUT cascades by slicing MTBDDs has a catch: the size and the width of MTBDD strongly depend on variable ordering. The problem of optimal variable ordering is unfortunately NP hard [5]. For some functions, the size of a BDD may vary between a linear to an exponential range depending upon the ordering of the variables. Thus the ROBDDs are canonical data structure for a fixed ordering only. The problem of ROBDD synthesis with optimal variable ordering for a complete Boolean function of n variables is computationally expensive as it can be done in time $O(n^2 3^n)$, [14]. Optimization of variable ordering for minimum size of ROBDD or MTBDD will very likely produce simultaneously a minimum BDD width (resulting in the most efficient LUT cascade).

Design of LUT cascades can start by determining the required number of rails k from the BDD width K. The value of K is known for sparse functions (Theorem 3), for weighted-sum functions [15], and some other function classes.

If BDD width K for the given function cannot be obtained from known expressions, one can get it from ROBDD for CF after minimizing its cost. This is, however, a separate problem. Recently, heuristic minimization algorithms have been proposed [8] that allow reduction of the WLDD size analogously as for BDDs. Next we will describe a co-synthesis of both MTBDD and LUT cascade for incompletely specified multiple-output Boolean functions. Let us note that sub-functions of incomplete function may themselves be also incomplete. A compatibility relation can be defined on the co-domain of such sub-functions: don't care (denoted by "x") is compatible with any value from Z_R .

The method is based on the bottom-up heuristic construction of BDDs by an iterative disjunctive decomposition. The illustrative incomplete function of 8 variables is given in Fig. 6. The map of the function at the top is sparsely populated by 16 function values (0 to F). For clarity don't care cells are left empty in tables, but otherwise are denoted by symbol "x" in the text.

Single-variable sub-functions can be created with respect to any variable. E.g. two vertically adjacent cells correspond to a sub-function of the first variable that attains alternate values 0 and 1 at even and odd rows (see e.g. [F, 8] in Fig.6). Using compatibility relation we can combine pairs [α,x] and [x,β] into a single sub-function [α,β]. Altogether nine sub-functions of the first variable are detected in the topmost table. The first decomposition step is described below the table; each sub-function is given a new ID code ([1,0]$\rightarrow$0, [2,7]$\rightarrow$1, [F,8]$\rightarrow$3, etc.), thereby removing the first variable from the function. A map of the new intermediate Boolean function of 7 variables is now created replacing sub-functions by new ID codes. This process repeats 8 times.

The MTBDD can now be created starting from root 0. Every assignment [a,b]$\rightarrow$ c, when reversed, specifies one decision node with input c and two outputs a and b controlled by the relevant variable. Assignments of the type [a,a]$\rightarrow$b, [a,x]$\rightarrow$c, [x,a]$\rightarrow$d do not represent decision nodes because the outputs are the same (or compatible); such a decision node degenerates to a wire. Going up from the root (a map of 0 variables) to the original map of 8 variables, the OBDD in Fig.7 is created. Usually BDDs have a root at the top, but we displayed the BDD upside down in order to keep the BDD structure in correlation with the sequence of map transformations in Fig.6. Nodes are labeled by intermediate sub-function values. Out of 46 assignments, 34 correspond to decision nodes and 12 to wires only.

In our example we did not care about variable ordering; the ordering was chosen more or less randomly. If we want to minimize the size of a BDD, the following heuristics can be used: do sub-function counting for all variables in each decomposition step and use in this step the variable with the minimum sub-function count K, what will ensure the minimum number of rails k, too. In case of ties, select the variable with the larger number of constant sub-functions. By intuition, the minimum count of sub-functions in one step may hopefully produce a minimum count of their pairs in the next step, and so on.

Note also that the above small example with maps of the original and intermediate functions was done only by hand for illustration. When we have sparse functions with several tens of variables represented by a list of defined points, all the processing is done automatically on these lists. The appropriate algorithm for such case is described in [6].

One layer of the MTBDD or more layers combined can be described by a LUT. For example LUT 4 in Fig. 8 is constructed from the layer of decision nodes adjacent to the leaves in Fig.7. Transformation of 9 function values to 16 ID codes is described by reversed assignments under the topmost table in Fig.6.

The whole BDD (Fig. 7a) is then described by 4 LUTs as shown in Fig. 7b. The LUT cascade is homogeneous, but generally the LUTs may have different size. However, sparse functions are typically implementable by homogeneous cascades, since the number of sub-functions (and therefore decision nodes) follows a pattern: rising – constant – dropping, [17].

Had we used a list of defined points with function values, there would be 39 items, 8 (input) + 4 (output) bits per item, 468 bits in total, i.e. half of the full function table with size 256 $\times$ 4 bits. Since we cannot order the items, we would have to use the linear search with up to 39 steps in the worst case.

On the other hand, if we use the LUT cascade according to Fig.7b, the capacity of all tables will be 4$\times$(32$\times$4) = 512 bits and only 4 steps (composed of read, append a value of a selected variable, add to the base address of the table to create a pointer) will do. This seems to be the best in speed and memory efficiency. Four tables may be implemented in memory as one table 32 $\times$ 16 bit with the correct output extracted from 16-bit word as needed. Additional flexibility is obtained with LUTs as they are combined together. For example with 2 tables 64 $\times$ 4 bits, the response will be 2-times faster. The total size of 2 and 4 LUTs remains the same, but 2 tables combined need 64 words in memory, 8 bits per word.

There are other heuristic approaches for MTBDD optimization. The basic operation for improving the variable ordering is the exchange of two adjacent variables. E.g. in sifting method [8] all position of a given variable in the given ordering are checked successively. The variable is then left in an optimal position with the lowest MTBDD size and process repeats for all variables. In dynamic variable ordering a window of m variables is moved over the n variables and in each position all permutations of the variables are considered [8]. The so called application specific variable ordering (ASVO) [8] uses structural properties of the problem instance for which the MTBDD has to be constructed.

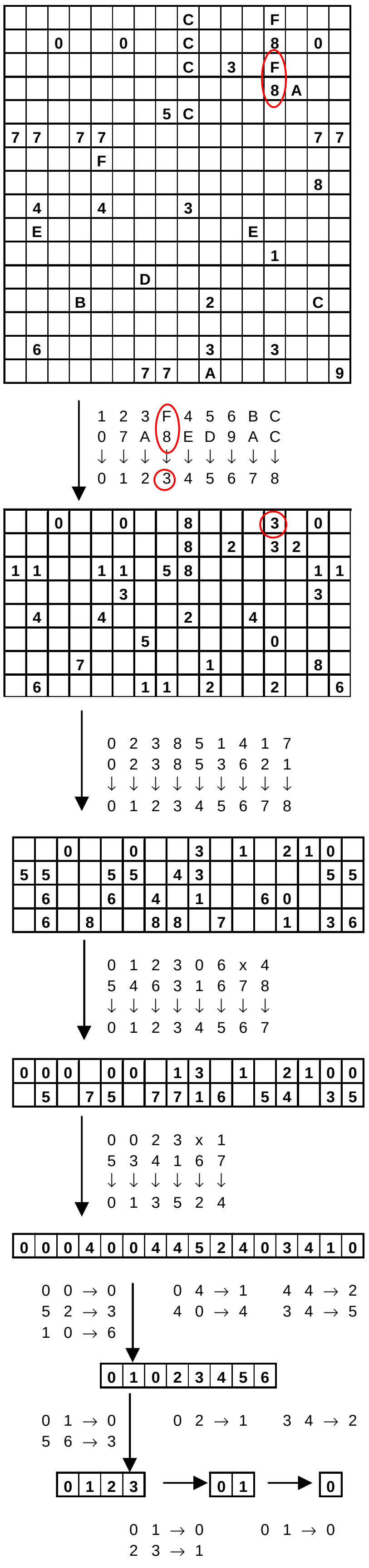
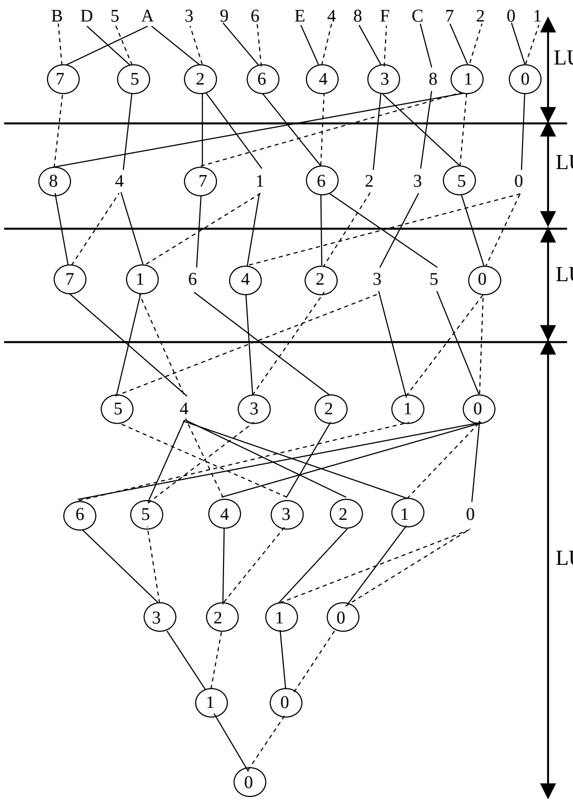
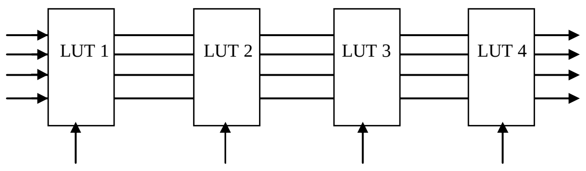


Fig. 6. Iterative decomposition (8 variables) (© IEEE 2007)



a)



b)

Fig.7. a) The ROBDD of the sample function of 8 variables (0 = - - - - - , 1 = ———)
b) LUT cascade (© IEEE 2007)

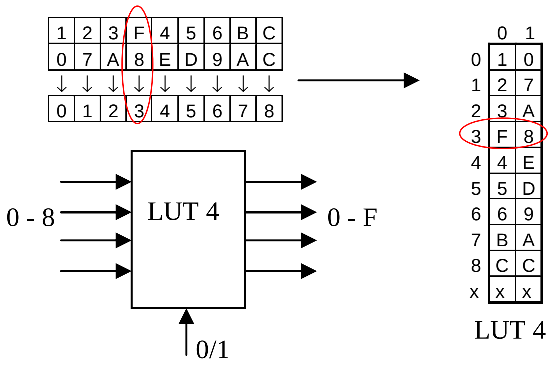


Fig.8. Construction of one LUT (© IEEE 2007)

Thorough comparison of all heuristic methods of optimization, as regards quality of results and an amount of the required execution time, remains still to be done.

VI. A CASE STUDY - MCS-51 MICROCONTROLLER FAMILY: PLA1 AND PLA2 IN SOFTWARE

Space and time efficiency of various configurations of LUT cascades obtained by computer-aided iterative decomposition have been tested on two PLAs used in the core of MCS-51 family of microcontrollers,

$$\text{PLA: } X \to Z_R,\ X \subset (Z_2)^n,\ Z_R \subset (Z_2)^r,$$

with parameters in Table 1. Both PLAs implement sparse (incomplete) Boolean functions, which are after minimization described by Boolean expressions (a list of expressions for PLA1 is given in appendix A). The number of terms in AND arrays are p = 31 and 53. The size in bytes gives memory space $r2^n$ required for storing full function tables.

TABLE 1.
PARAMETERS OF PLA1 AND PLA2

	n	r	p	|X|	size [B]
PLA1	13	8	31	175	8192
PLA2	11	8	53	632	2048

Iterative decomposition used the selection of those two variables at a time that produced the minimum number of sub-functions. Not too large size of the problem allowed still an exhaustive search – on the Pentium-based PC it took tens of seconds. The PLA1 was implemented by the cascade of 6 cells, Fig. 9a, with the total size of LUTs only 1792 bits. That is reduction by factor of 36. The size of LUTs is not uniform and evaluation would take 6 table look-ups. We can make it faster and more uniform by combining 6 cells into 3 as shown in Fig. 9b. All sub-functions are counted (results given in {integer}), coded and communicated between cells; function values are outputs from the last cell only. The total size of all LUTs is then 2816 bits; if the size of computer word w is known, further optimization can be done to minimize the total memory space in bytes occupied by all 3 LUTs.

As far as PLA2 is concerned, computer-generated cascades are shown in Fig. 10. The cascade at Fig. 10b is obtained from the cascade a) by merging first two LUTs. The capacity of LUTs is 3264 and 3456 bits, respectively. The evaluation speed is given by 4 or 3 table look-ups.

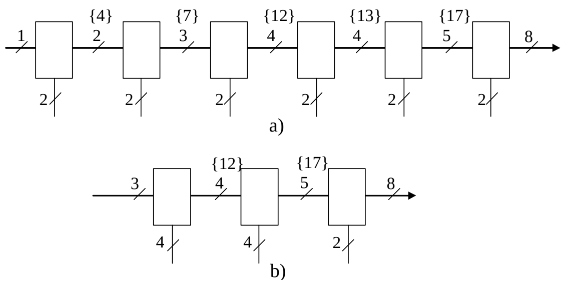


Fig. 9. Two cellular cascade implementations of PLA1 (© IEEE 2007)

We can also use an optimization technique based on creating groups of outputs and design a separate LUT cascade for each group. Multiple cascades may be narrower and shorter than the original one and can save some memory space, but number of table lookups will be always greater. A problem how to split output variables optimally into groups is solved typically by heuristic methods [18], [16]. In case of PLA2 we will split output variables intuitively into two halves and then decompose them separately. The result is shown at Fig. 11. The size of LUTs is reduced to 1200 bits only, but the speed is reduced also. Eight table look-ups are needed and can be done on one CPU core in 8 steps sequentially or on a 2-core processor concurrently in 4 steps.

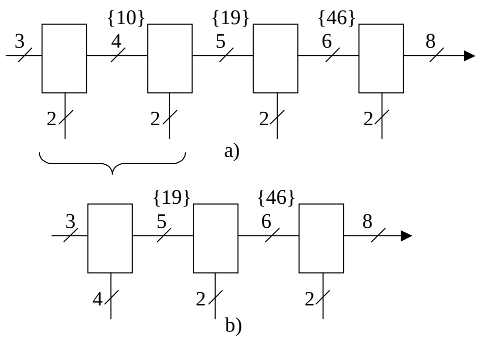


Fig.10. Cascade of 4 or 3 cells for PLA2 (© IEEE 2007)

The case study of PLA1 and PLA2 offered the size of data structures and speed of evaluation as given in Table 2. The data in the table are valid under the assumptions:

- size is in bits, the length of a computer word is not considered;
- steps may have different duration at PLA emulation and LUT cascade processing (mask load + bitwise logical operation vs table look-ups).

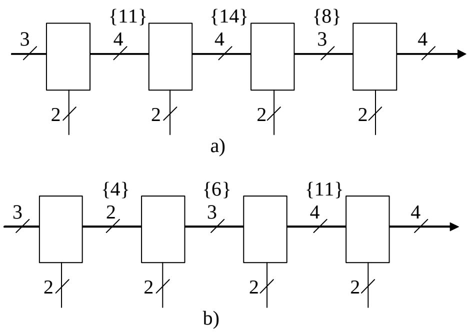


Fig. 11. Two parallel cascades implementing PLA2.

TABLE 2.
SOFTWARE IMPLEMENTATIONS OF PLA1 AND PLA2

	PLA emulation AND + OR matrix		LUT cascades	
	size bits	steps	size bits	steps
PLA1	1054	13 + 8	1792	6
PLA1	1054	31 + 8	2816	3
PLA2	1590	11 + 8	3456	3
PLA2	1590	53 + 8	1200	8

VII. Microprogrammed controller with multi-way branching

Evaluation of Boolean functions at the firmware level can also benefit from the LUT cascade paradigm. By making use of a hardware micro-engine with support for multi-way branching, we can speed up evaluation of Boolean functions against a general purpose CPU core. A suitable architecture of a micro-engine similar to [19] is depicted in Fig.12.

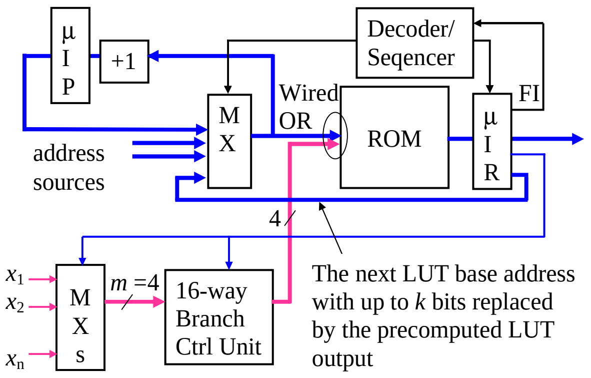


Fig.12. Micro-programmed controller architecture with multi-way branching

There are three microinstructions formats (FI = Format Indicator):

FI = 1: state output (control signals), μIP := μIP+1

FI = 10: MXs and BCU control, jump to an address specified in micro-instruction modified by BCU

FI = 11: conditional output, jump to an address specified in micro-instruction (no modification).

The second format is for all kinds of jumps to an address specified in micro-instruction. This 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). The task of this unit is to shift active inputs, selected by a 4-bit mask, to the lowest positions of the 4-bit output vector. This vector is then wire-ORed with the address obtained from the micro-instruction. If there are more external variables, the LUT cascade paradigm is used. LUT output contains not only the rail variables, but the whole next LUT base address modified by k rail variables in proper positions.

We will illustrate rewriting a general multi-way branch microinstruction into a micro-program. The multi-way branch has the same structure as a switch. Let us have the statement

```
S0:  if  F = 0 then v0 exit S0
     if  F = 1 then v1exit S1
     if  F = 2 then v2 exit S1
     if  F = 3 then v2 exit S2
     if  F = 4 then v3 exit S3
     else don´t care;                    (13)
```

Si´s are state labels, vj´s are conditional output vectors, $F(A,B,C,D)$: $X \rightarrow Z_5$, $X \subset (Z_2)^4$ is an incomplete multiple-output Boolean function, its map is in Fig. 13a.

$F(A,B,C,D)$ $CD \longrightarrow$; $AB \downarrow$

AB \ CD	00	01	10	11
00	0	1	2	2
01	x	2	2	2
10	x	4	0	1
11	4	x	2	3

a)

```
S0 exit L@ABCD
...
L@0000 v0 exit S0
L@0001 v1 exit S1
...
...
L@1110 v2 exit S1
L@1111 v2 exit S2
```

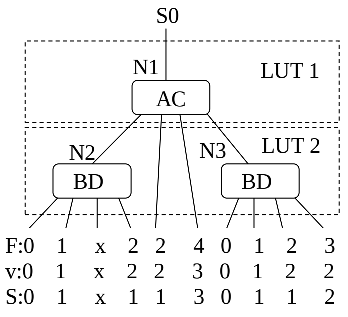
b)

Fig.13. The map of a sample function (a) and a symbolic dispatch table in the micro-program (b)

The switch statement (13) describes a transition from present state $S0$ to one of next states $S0$ to $S3$ depending on the values of 4 external variables A, B, C and D. During the transition a certain conditional output vector vj is generated.

If the speed of the micro-engine is the utmost priority, we should do the testing of external variables in one step. The 16-way branch is then translated to the dispatch table in Fig.13b. Replacement of 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.

If saving in hardware (chip area) is more important than overall speed, we can test variables A, B, C and D in groups of two. The optimum MTBDD found by the method described in Section V is shown in Fig. 14, together with the symbolic micro-program derived from it. It can be seen, that the second LUT is only partial as two sub-functions of two variables A, C are constants (2 and 4). Control store capacity is almost half of the capacity in the previous case and the BCU can be eliminated.



a)

```
S0 exit N1@AC
N1@00 exit N2@BD
N1@01 v2 exit S1
N1@10 v3 exit S3
N1@11 exit N3@BD
N2@00 v0 exit S0
N2@01 v1 exit S1
N2@10 -
N2@11 v2 exit S1
N3@00 v0 exit S0
N3@01 v1 exit S1
N3@10 v2 exit S1
N3@11 v2 exit S2
```

b)

Fig. 14. LUT cascade (a) and the symbolic micro-program (b) for a multiway branch example

As the last example we shall consider evaluation of the following sparse Boolean function of 16 variables: it attains the value 1 if the given 6-bit string is detected anywhere within an input string of 16 Boolean values; otherwise the function has the value 0.

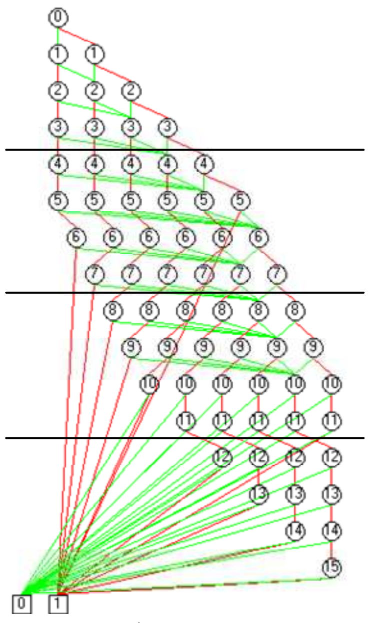
Since the string of 6 consecutive values of variables may be located in 11 positions (we do not assume that the pattern wraps around), we can specify the function by 11 words of 16 ternary digits (0, 1, x). The CPU evaluation of this function by logarithmic search is not possible and we have to step through these words sequentially. In the worst case it may take 11 steps.

We can do much faster with LUTs, though. First the ROBDD of this function may be obtained using the applet [12], since the Boolean expression with 11 min-terms, each with 6 literals, is easy to write down (for a pattern of six 1´s):

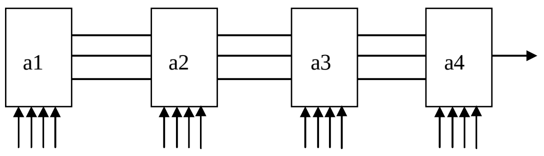
a1*a2*a3*a4*a5*a6+a2*a3*a4*a5*a6*a7+a3*a4*a5*a6*a7*a8+a4*a5*a6*a7*a8*a9+a5*a6*a7*a8*a9*a10+a6*a7*a8*a9*a10*a11+a7*a8*a9*a10*a11*a12+a8*a9*a10*a11*a12*a13+a9*a10*a11*a12*a13*a14+a10*a11*a12*a13*a14*a15+a11*a12*a13*a14*a15*a16 (14)

The ROBDD is in Fig. 15a, from which an optimal size and count of LUTs can be determined. We have used 4 LUTs with 3 rails and 4 vertical inputs for the target micro-controller architecture in Fig.12.

The micro-program would consist of $16 + 3 \times (8 \times 16) = 400$ jump microinstructions, but only 4 of them would be executed for the given input vector. Execution time (of 4 microinstructions) will be somewhere between response of software realization (4 table lookups) and a hardware LUT cascade.



a)



b)

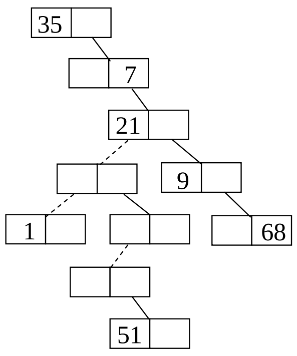
Fig. 15. The ROBDD (a) and LUT cascade detecting 6-bit string in 16 bits (b)

To complete the discussion on the pattern detection example, let us note that LUT cascade can be constructed for any number of input variables. The width of cascade remains the same and the BDD has a repetitive nature. We can think of it as of repeated evaluation of a Boolean function that depends on small number of local variables defined by a window that moves along the data stream.

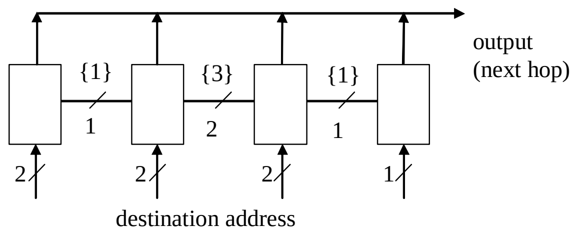
Important application of pattern matching is the IP address lookup, one of the primary functions of a router, and often also a significant performance bottleneck. An Internet router table is a set of tuples of the form (p, a), where p is a binary string whose length is at most n (n = 32 for IPv4 destination addresses and n = 128 for IPv6), and a is an output link (or next hop). When a packet with destination address A arrives at a router, we are to find the pair (p, a) in the router table for which p is a longest matching-prefix of A (i.e., p is a prefix of A and there is no longer prefix q of A such that (q, b) is in the table). Once this pair is determined, the packet is sent to output link a. The speed at which the router can route packets is limited by the time it takes to perform this table lookup for each packet.

Prefixes	Next hop
0*	35
11*	7
110*	21
1110*	9
11000*	1
11111*	68
1101010*	51

a)



b)



c)

Fig.16. Prefix table (a), corresponding 1-bit trie (b) and the LUT cascade with intermediate outputs (c)

An example of prefix table is in Fig.16a. It is in fact fully specified multiple-output Boolean function with 2^{n-1}, 2^{n-2},…, and 2 input vectors (destination addresses) producing the same next hop. The tree-like MTBDD of such type of function is called trie, Fig.16b [20]. Nodes of a trie contain two element fields, each element field has the components child and data (output). Branching is done based on the bits in the search key. A left-element child branch is followed if the ith bit of the search key is 0; otherwise a right-element child branch is followed.

Prefix output evaluation is conducted by using the IP address bits to traverse the trie, starting with the most significant bit of the address. To speed up this searching process, multiple bits of the destination address are compared simultaneously in a multibit trie. m levels of the binary trie are combined into single LUT what reduces the number of memory accesses needed to perform the whole IP address lookup; *m* is called stride. In Fig. 16c stride *m* = 2. Even though the intermediate outputs are produced often earlier in the cascade, they are always forwarded to the last LUT, [21]. For high speed packet forwarding the speed of micro-engine in Fig. 12 is not sufficient. Two hardware-based solutions are available: those that involve the use of a CAM or that use synchronous LUT pipeline (on ASIC or FPGA) [21], [22]. The CAM can be implemented by a LUT cascade anyway [23]. In both the cases one additional complication has to be addressed, namely frequent modifications of the routing table. Some additional hardware is required for this functionality and programmable LUT cascades are thus obtained [24].

VIII. Conclusion

There is no single software evaluation method optimal for all Boolean functions. Complexity of functions that can appear in embedded systems varies a great deal and so do their space and time requirements in various evaluation techniques.

Even though the very narrow analysis done above cannot be taken as convincing, certain conclusions for engineering practice can be drawn from it, if the fast and memory efficient evaluation of sparse Boolean functions $F_n : X \rightarrow Z_R$ of several tens of variables is the main concern.

1. If the set $X \subset Z_2^n$ contains only a small number of elements, e.g. when the function is specified by DNF with low tens of min-terms, the search in the ordered list of min-terms can be very effective solution.

2. If $X \subset \{0,1,x\}^n$, sequential TCAM emulation may be too slow as it takes | X | steps. LUT cascades are a good solution. Generally speaking, every sparse function can be implemented as a LUT cascade.

3. OBDDs or ROBDDs may be useful for checking equivalence between two implementations or for formal verification [1], but they are less useful for evaluation purposes in both speed and memory consumption.

4. LUTs obtained from MTBDDs seem to be a very good and effective data structure and should always be considered for evaluation of Boolean functions. They are flexible in making trade-offs between response time and memory consumption. LUT cascades implemented directly in hardware can support asynchronous or synchronous pipeline processing [25]. Otherwise, in case of software implementation, several LUTs can be compacted into one block of memory words. The evaluation then reduces to a short chain of indirect memory accesses.

Future research will be oriented to applications of the iterative decomposition and LUT cascades in the area of secure and safe hardware. The intention is to decompose large systems of sparse Boolean functions of many variables, which appear in this area, into LUT cascades with the aid of evolutionary optimization techniques. A generalization to LUT networks will also be studied.

Appendix A
Programmable logic array PLA1 in MCS-51 microcontroller family

Legend: ! = logical negation, * = logical AND,
+ = logical OR

PLA1
Inputs: A, B, C, D, E, F, G, H, I, J, K, L, M
Outputs: SO, CS, BL, NL, V1, V3, V4, V5

S0 = !A*!G*!I*J*M+A*!B*!I*J*M+A*F*!I*M

CS = !A*!B*D*!E*!F*!G*!H*!I*!K*!L*M +
A*B*!E*!F*!G*!H*!I*!J!K*!L*!M + !A*!E*!I*M +
!E*!I*J*M+!D*!I*M

BL = !B*E*!F*!G*!H*!I*!J*!K*!L +
!B*C*!D*!H*!I*!J*M + !B*D*E*!H*!I*!J*M +
!D*!I*!J*K*M + !A*!G*!I*J*M + E*H*!I*!L*M
+C*!D*G*!I*M + !A*F*!I*M + G*!I*K*M +
E*G*!I*M

NL = !B*E*!F*!G*!H*!I*!J*!K*!L + C*!D*!H*!I*L*M
+ !D*!I*!J*K*M + !A*!G*!I*J*M + D*E*!N*!I*M
+ !A*F*!I*M + E*!I*!L*M + G*!I*K*M

V1 = !A*!G*!I*J*M + C*!D*F*!I*M + A*!B*!I*J*M +
!A*F*!I*M + F*!I*K*M + E*F*!I*M

V3 = !B*!C*!D*E*!F*!G*!H*!I*!J*!K*!L +
!B*!G*!I*J*K*M + !D*!I*!J*K*M + B*C*!I*K*M

V4 = !B*C*!D*E*!F*!G*!H*!I*!J*!K*!L +
!B*D*E*!F*!G*!H*I*!J*!K*!L*M +
!A*!G*!I*J*L*M + C*!D*!H*!I*L*M +
!A*F*!I*L*M + C*!D*H*!I*M +D*E*!I*L*M

V5 = !B*D*E*!F*!G*!H*I*!J*!K*!L*M +
!B*E*!F*!G*!H*!I*!J*!K*!L + C*!D*!H*!I*L*M
+!D*!I*!J*K*M + !A*!G*!I*J*M + C*!D*H*!I*M
+ A*!B*!I*J*M + D*E*!I*L*M + !A*F*!I*M +
E*!I*!L*M

Acknowledgment

This research has been carried out under the financial support of the research grants GA102/07/0850 “Design and hardware implementation of a patent-invention machine”, GA102/05/0467 “Architectures of Embedded Systems Networks”, Grant Agency of Czech Republic, and “Security-Oriented Research in Information Technology”, MSM 0021630528.

References

[1] B.M. Moret: Decision Trees and Diagrams. Computing Surveys, Vol.14, No.4, Dec. 1982, pp. 593-623.

[2] F. D. Petruzella: *Programmable Logic Controllers,* McGraw Hill Science/Engineering/Math, 2004.

[3] R. Sosic, J. Gu, and R. Johnson. “The Unison algorithm: Fast evaluation of Boolean expressions”. *ACM Transactions on Design Automation of Electronic Systems,* 1(4): pp. 456--477, Oct. 1996.

[4] T. Sasao, Y. Iguchi, M. Matsuura, "LUT cascades and emulators for realizations of logic functions," *RM2005,* Tokyo, Japan, Sept. 5 - Sept. 6, 2005, pp.63-70.

[5] B. Bollig, I. Wegener, “Improving the Variable Ordering of OBDDs Is NP-Complete”. *IEEE Transactions on Computers*, 45(9):993—1002, September 1996.

[6] V. Dvořák: An optimization technique for ordered (binary) decision diagrams, In: *Proceedings of the 6th Annual European Computer Conference CompEuro' 92*, Hague, NL, 1992, pp. 1-4, ISBN 0-8186-2760

[7] A. Mishchenko, T. Sasao: Logic Synthesis of LUT Cascades with Limited Rails - A Direct Implementation of Multi-Output Functions. *Technical report of IEICE*, The Institute of Electronics, Information and Communication Engineers Vol.102, No.476(20021121) pp. 103-108. VLD2002-99, ISSN:09135685.

[8] R. Drechsler, B. Becker, *Binary Decision Diagrams - Theory and Implementation*. Springer 1998.

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

[10] V. Dvořák: Bounds on Size of Decision Diagrams, JUCS - *The Journal of Universal Computer Science* (JUCS), Vol..3, pp. 2-23, 1997.

[11] V**.** Dvořák: A cascade implementation of digital systems. In: *Microprocessing and Microprogramming* (North-Holland), Vol. 29, No. 1, 1990, pp. 151-163, ISSN 0165-6074.

[12] University of Hamburg, Department of Informatics, http://tams-www.informatik.uni-hamburg.de/applets

[13] W. Stallings, *Computer Organization and Architecture*, Sixth Edition, Prentice Hall, 2005.

[14] S.J.Friedman, K.J.Supowit: Finding the Optimal Variable Ordering for Binary Decision Diagrams. *IEEE Transactions on Computers*, Vol. 39, No. 5, pp.710-713, May 1990.

[15] T. Sasao: Analysis and Synthesis of Weighted-Sum Functions. *Proc. Of the Int. Workshop on Logic and Synthesis*, Lake Arrowhead, CA, USA, June 8-10, 2005, pp.455-462.

[16] T.Sasao, M.Matsuura: A Method to Decompose Multiple-Output Logic Functions. *Proc. of the 41st Design Automation Conference*, San Diego, CA, USA, June 2-6, 2004, pp.428-433.

[17] V. Dvořák: *Decomposition Theory with Applications in Programmable Digital Systems*. A thesis required for Doctor of Sciences degree. Faculty of El. Engineering, Technical University of Brno, May 1989. (224 pages, in Czech).

[18] R. Drechsler, M. Herbstritt, B. Becker Grouping heuristics for word-level decision diagrams. *Proceedings of the 1999 IEEE International Symposium on Circuits and System* ISCAS '99, pp. 411--415.

[19] V. Dvořák: Microsequencer architecture supporting arbitrary branching up to 2^m targets, *Computer Architecture News*, IEEE Publ., US, March 1990, pp. 9-16.

[20] K.S. Kim, S. Sahni: Efficient Construction of Pipelined Multibit-Trie Router-Tables. *IEEE/ACM Transactions on Networking* (TON), Volume 11 , Issue 4 , August 2003, pp. 650 – 662. ISSN:1063-6692

[21] T. Henriksson, I.Verbauwhede: Fast IP Lookup Engine for SoC Integration. *Proc. of IEEE Design and Diagnostics of Electronic Ciruits and Systems Workshop*, April 17-19, 2002, Brno, Czech Republic, pp.200-210.

[22] D. E. Taylor, J. W. Lockwood, T. S. Sproull, J. S. Turner, D. B. Parlour: Scalable IP Lookup for Programmable Routers. *Tech. Rep. WUCS-01-33*, Dept. of Computer Science, Applied Research Lab, Washington University, 2001.

[23] T.Sasao and J. T. Butler, "Implementation of multiple-valued CAM functions by LUT cascades," *ISMVL-2006*, Singapore, May 17-20, 2006.

[24] H. Nakahara, T. Sasao, M.Matsuura: A CAM Emulator Using Look-Up Table Cascades. *14th Reconfigurable Architectures Workshop* RAW 2007, March 2007, Long Beach California, USA. CD-ROM RAW-9-paper-2.

[25] 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*, April 20-22, 2005, Yokohama, Japan.

Vaclav Dvorak obtained M.Sc. degree in El. Engineering and Ph.D. degree in Applied Cybernetics from Brno University of Technology, Czech Republic, in 1963 and 1968. He was awarded a distinguished DrSc degree in Computer Science and Engineering in 1990.

Since 1963 he was 10 years with the Research Institute of Mathematical Machines Prague. Then he joined Brno University of Technology, Faculty of Information Technology, as a research associate and later as Associate and Full Professor. The major field of his interest has been computer hardware and architecture. He interleaved the work at the home university with acting as visiting scientist, lecturer and professor at a number of foreign institutions, over 8 years in all. (Canada, Malta, Libya, New Zealand, Australia, Tenerife-Spain). His research is recently oriented into application specific and parallel architectures.

Prof. Dvorak is a member of Computer Society and IEEE, a member of the Scientific Board of the Faculty of Information Technology, committees for Bc, MSc and Ph.D. studies in Information Technology and a member of JUCS and JEE Editorial Boards.