# Encoding of Boolean Functions and Its Application to LUT Cascade Synthesis

**Alan Mishchenko**
Department of ECE
Portland State University
Portland, OR 97207, USA
alanmi@ece.pdx.edu

**Tsutomu Sasao**
Center for Microelectronic Systems and Department of CSE
Kyushu Institute of Technology
Iizuka, Fukuoka, 820-8502 JAPAN
sasao@cse.kyutech.ac.jp

## Abstract[1]

*The problem of encoding arises in several areas of logic synthesis. Due to the nature of this problem, it is often difficult to systematically explore the space of all feasible encodings in order to find an optimal one.*

*In this paper, we show that when the objects to be encoded are Boolean functions, it is possible to formulate and solve the problem optimally. We present a general approach to the encoding problem with one or more code-bit functions having some desirable properties. The method allows for an efficient implementation using branch-and-bound procedure coupled with specialized BDD operators.*

*The proposed approach was used to synthesize look-up table (LUT) cascades implementing Boolean functions. Experimental results show that it finds optimal solutions for complex encoding problems in less than a second of CPU time.*

## 1 Introduction

A binary encoding can be represented by a mapping from the set of objects into the set of all subsets of minterms of the Boolean space, $B^w = \{0,1\}^w$, where $\mu$ is the number of objects and $w \geq log_2 \mu$. An additional requirement is that the subsets of minterms used for the codes do not overlap.

The encoding problems have been extensively studied for state assignment of finite state machines (FSMs), in particular, in asynchronous synthesis [15], generation of minimum-area PLAs [6], and design of state-event systems [3]. In these applications, the objects to be encoded are FSM states.

In several other areas, the objects to be encoded are Boolean functions. These areas include Ashenhurst-Curtis decomposition of binary and multi-valued relations [10], functional decomposition for FPGAs [5], and binary encoding of multi-valued networks [3].

Previous approaches to encoding Boolean functions relied on encoding with input and output constraints from state assignment of FSMs [8][3], the use of heuristics [10][1], or Boolean satisfiability [9][5]. Although the latter two approaches tend to work better because they search a larger solution space, they are often slow due to large representations for *all* encoding choices.

[1] The first author has been partially supported by a research grant from Intel Corporation.

This paper continues research started in [12] and presents a general solution to encoding of Boolean functions with the following optimality criteria: (1) the resulting code-bit functions have small support sizes, or (2) the resulting code-bit functions belong to a certain class of functions. The first requirement is typical in the decomposition for FPGAs because functions depending on a single variable are implemented as a wire without a LUT. The second type occurs in technology-dependent logic synthesis when the Boolean network is decomposed/mapped into a standard-cell library.

To experimentally evaluate the proposed encoding algorithm, we apply it to synthesis of the LUT cascade representation of Boolean functions [11]. Experimental results show that the new encoding algorithm significantly reduces the number of LUTs compared to random encoding. A similar study has been recently performed using fast heuristic encoding algorithm [2].

The rest of the paper is organized as follows. Section 2 defines the encoding problem and introduces the notation. Section 3 derives and compares the number of strict and non-strict encodings. Section 4 presents the theoretical background of the paper. Section 5 discusses the efficient implementation of the algorithm using branch-and-bound search and specialized BDD operators. Section 6 shows the application of the encoding algorithm to LUT cascade synthesis. Section 7 gives experimental results. Section 8 draws conclusions and outlines future work.

## 2 Definitions

Given a set of objects $\{o_i\}$, $1 \leq i \leq \mu$, and a set of Boolean variables, $\{z_j\}$, $1 \leq j \leq w$, $w \geq \lceil log_2 \mu \rceil$, a *strict encoding* is an assignment of a unique minterm $m_i(z)$ to each object $o_i$. A *non-strict encoding* is an assignment of a unique function $c_i(z)$ to each object $o_i$ in such a way that the functions $c_i(z)$ are pair-wise disjoint. The encodings discussed in this paper are always non-strict unless stated otherwise.

Parameter $w$ is *code length*. Variables $z_j$ are *code bits*. Minterm $m_i(z)$, or function $c_i(z)$, are the *code* of object $o_i$.

A strict encoding is *straight-binary* (*natural*) if the codes are minterms representing integer indices of the objects. The minimum length of the code needed to encode $\mu$ objects is $\lceil log_2 \mu \rceil$. An encoding with $w = \lceil log_2 \mu \rceil$ bits is a *minimum-length* (or *logarithmic*) encoding.

*Example 1*. Table 1 gives examples of a strict and a non-strict encodings of objects $\{o_1, o_2, o_3\}$ using binary variables $\{z_1, z_2\}$.

The strict encoding is a straight-binary encoding with the unused code $\bar{z}_1\bar{z}_2$.

*Table 1. Examples of strict and non-strict encodings.*

| Objects | Strict encoding | Non-string encoding |
|---|---|---|
| $o_1$ | $\bar{z}_1 z_2$ | $\bar{z}_1\bar{z}_2$ |
| $o_2$ | $z_1\bar{z}_2$ | $\bar{z}_1 z_2 \vee z_1\bar{z}_2$ |
| $o_3$ | $z_1 z_2$ | $z_1 z_2$ |

In this paper, we consider the encoding problem when the objects to be encoded are non-overlapping (disjoint) Boolean functions, $f_i(x)$, $1 \le i \le \mu$. When this problem arises in functional decomposition, the parameter $\mu$ is called *column multiplicity*.

An encoding of a set of functions can be represented by the *encoding relation*, $C(x, z)$, mapping the domain of the functions into the domain of the code variables. The encoding relation is the sum of the products of functions $f_i(x)$ by their codes $c_i(z)$:

$$C(x, z) = \bigvee_{i=1}^{n} [\, f_i(x) \wedge c(z)]$$

The *code-bit functions* (or simply, *code functions*) express each bit of the code in terms of support variables of $f_i(x)$. The code-bit functions are derived by cofactoring $C(x, z)$. The $j$-th code-bit function $z_j(x)$ is the positive cofactor of $C(x, z)$ w.r.t. $z_j = 1$, with the remaining code bit variables existentially quantified:

$$z_j(x) = \exists_z [C(x, z) \,|\, z_j{=}1].$$

*Example 2.* Let objects $\{o_1, o_2, o_3\}$ in Example 1 be functions $\{ x_1x_2,\ x_1\bar{x}_2,\ \bar{x}_1 \}$. For the strict encoding shown in Table 1, the encoding relation and code-bit functions are:

$$C(x, z) = x_1x_2\ \bar{z}_1z_2 \vee x_1\bar{x}_2\ z_1\bar{z}_2 \vee \bar{x}_1\ z_1z_2,$$

$$z_1 = \bar{x}_1 \vee \bar{x}_2,\ z_2 = \bar{x}_1 \vee x_2.$$

Given a set of functions $f_i(x)$, $1 \le i \le \mu$, and a function $g(x)$, which may belong to the set, the number of functions in $f_i(x)$ that overlap with $g(x)$ is denoted $Count(f_i, g)$.

Consider functions $f_i(x)$, $1 \le i \le 3$, in Example 2 and the function $g(x) = x_2$. $Count(f_i, g) = 2$, because $g(x) = x_2$ does not overlap with $f_2 = x_1\bar{x}_2$, but overlaps with $f_1 = x_1x_2$ and $f_3 = \bar{x}_1$.

## 3 Number of Encodings

This section considers the number, $N_E(\mu, w)$, of different encodings of the set of $\mu$ objects using $w$-bit codes.

If $\mu = 2^w$, the number of encodings is equal to the number of different permutations of $\mu$ minterms, $N_E(\mu, w) = \mu!$.

If $\mu < 2^w$, for each out of $\mu!$ encodings, there are $\dfrac{2^w!}{(2^w - \mu)!\,\mu!}$ ways of choosing minterms used in the codes, which yields the total of $N_E(\mu, w) = \dfrac{2^w!}{(2^w - \mu)!}$.

In the case of non-strict encodings, the number is even larger. Let us assign $\mu$ minterms to $\mu$ functions, which gives $\mu!$ encodings. The remaining $2^w - \mu$ minterms can either be assigned to one of the functions or left unused. This gives an upper bound on the number of different non-strict encodings:

$$N_E(\mu, w) \le \mu!(\mu + 1)^{2^w - \mu}.$$

The formulas are illustrated in Table 2. The table shows that the number of different encodings is extremely large.

*Table 2. The number of strict and non-strict encodings.*

| $\mu$ | $w$ | Strict encodings | Non-strict encodings |
|---|---|---|---|
| 3 | 2 | 24 | 96 |
| 5 | 3 | 6720 | 25920 |
| 10 | 4 | $2.9 \cdot 10^{10}$ | $6.4 \cdot 10^{12}$ |
| 20 | 5 | $5.5 \cdot 10^{26}$ | $1.8 \cdot 10^{34}$ |
| 40 | 6 | $2.0 \cdot 10^{65}$ | $4.2 \cdot 10^{86}$ |
| 100 | 7 | $1.2 \cdot 10^{186}$ | $1.2 \cdot 10^{214}$ |

## 4 Theoretical Background

The encoding algorithm is based on the following theorem.

**Theorem 1.** Let $f_i(x)$, $1 \le i \le \mu$, be Boolean functions and the encoding variables be $z_j$, $1 \le j \le w$, $w \ge \lceil log_2 \mu \rceil$. An encoding of function $f_i(x)$ using variables $z_j$ with the given code-bit function $g(x)$ exists if and only if

$$Count(f_i, g) \le 2^{w-1} \text{ and } Count(f_i, \bar{g}) \le 2^{w-1}.$$

**Proof.** (*Sufficiency*) Suppose a non-strict encoding with function $g(x)$ exists. Setting the code-bit to 1 gives $Count(f_i, g)$ functions to be encoded by the remaining $w$-1 bits. It means that there are no more than $2^{w-1}$ functions, that is, $Count(f_i, g) \le 2^{w-1}$.

Similarly, it is proved that $Count(f_i, \bar{g}) \le 2^{w-1}$.

(*Necessity*) If the above limits on the number of overlapping functions are true, then each group of functions can be encoded using $w$-1 encoding bits. Taken together with the code-bit of $g(x)$, these bits create a non-strict encoding. *Q. E. D.*

A theoretical result similar to Theorem 1 was formulated in [5] (Property 1) without proof and used to implicitly enumerate support-reducing encoding choices. In this paper, we consider another application of the same result.

*Example 3.* Consider five non-overlapping functions depending on four variables. Figure 1 shows the Karnaugh map. The minterms in the map are labeled by the functions covering these minterms. Let us encode these five functions using variables $\{z_1, z_2, z_3\}$ and assume that the given code-bit function is $g(x) = x_1$.

| $x_3x_4$ \ $x_1x_2$ | $g(x)=0$ 00 | 01 | $g(x)=1$ 11 | 01 |
|---|---|---|---|---|
| 00 | $f_5$ | $f_2$ | $f_1$ | $f_5$ |
| 01 | $f_5$ | $f_2$ | $f_1$ | $f_3$ |
| 11 | $f_4$ | $f_1$ | $f_1$ | $f_3$ |
| 10 | $f_4$ | $f_1$ | $f_1$ | $f_4$ |

*Figure 1. Example illustrating Theorem 1.*

The code-bit function $g(x)$ splits the Boolean space into two parts (to the left and to the right of the double line). In each part, there are minterms of four different functions ($\{f_1, f_2, f_4, f_5\}$ on the left, $\{f_1, f_3, f_4, f_5\}$ on the right). These four minterms can be encoded using two remaning two bits.

Suppose the functions on the left and on the right of the double line are encoded as shown in Table 3. (The dash indicates that a function is missing in this part.) The resulting encoding is shown in column "Code". Note that this encoding is non-strict because the codes of $f_1$, $f_4$, and $f_5$ contain more than one minterm.

*Table 3. Encoding of functions in Example 3.*

| Function | Left part ($z_1 = 0$) | Right part ($z_1 = 1$) | Code |
|---|---|---|---|
| $f_1$ | $\bar{z}_2\bar{z}_3$ | $\bar{z}_2\bar{z}_3$ | $\bar{z}_2\bar{z}_3$ |
| $f_2$ | $\bar{z}_2 z_3$ | - | $\bar{z}_1\bar{z}_2 z_3$ |
| $f_3$ | - | $\bar{z}_2 z_3$ | $z_1\bar{z}_2 z_3$ |
| $f_4$ | $z_2\bar{z}_3$ | $z_2 z_3$ | $\bar{z}_1 z_2\bar{z}_3 \vee z_1 z_2 z_3$ |
| $f_5$ | $z_2 z_3$ | $z_2\bar{z}_3$ | $\bar{z}_1 z_2 z_3 \vee z_1 z_2\bar{z}_3$ |

# 5 Implementation Issues

## 5.1 Considering Multiple Code-Bits

In practical applications it is often important to find an encoding, in which not one, but the largest possible number of code-bit functions are one-variable functions. To achieve this, Theorem 1 is applied iteratively, as long as a feasible non-strict encoding exists.

Suppose we applied Theorem 1 for the first time and found the first code-bit function equal to the input variable $x_{i1}$. Next, the functions are cofactored w.r.t. variable $x_{i1}$, and Theorem 1 is applied to the set of positive cofactors and the set of negative cofactors. If a feasible encoding with a simple code-bit function exists for both cofactors, we continue searching for the next code bit, and so on.

The iterative computation scheme can be realized as a branch-and-bound search. At each branching point, we try all input variables that are not used as simple code-bit function. If a variable can be a code-bit function, we branch to the next level. If on some level, we tested all input variables and none of them worked, the branch-and-bound procedure backtracks to the previous level.

The runtime of the branch-and-bound procedure can be significantly reduced using upper and lower bounds updated dynamically as computation proceeds.

The branch-and-bound algorithm can solve the encoding problem *exactly* in the following sense. If there exists an encoding with $u$ out of $w$ code-bits, $u \leq w$, represented by the simple functions (functions depending on one variable, or functions implementing library gates), this encoding is found, and $u$ is guaranteed to have the largest value.

## 5.2 Encoding Steps

The encoding algorithm takes a set of functions to be encoded, $f_i(x)$, and a set of preferred code-bit functions, $g_k(x)$. The algorithm tries to find a feasible encoding with the largest number of code-bit functions in the given set. The resulting encoding is in the form of the encoding relation.

The following steps are performed repeatedly in the above branch-and-bound procedure:

(1) Count the number of functions in the set $f_i(x)$ that are overlapping with the function $g(x)$, $Count(f_i, g)$.
(2) Cofactor all the functions belonging to a set w.r.t. a variable.
(3) Extract the individual codes from the encoding relation.

The naïve implementation iterates through all the functions in the set to perform operations (1) and (2). This leads to a noticable slow-down when the number of functions is large (say, 1000).

In a more efficient implementation, the set of functions is represented by the encoding relation assuming a natural encoding of functions. This representation reduces operation (1) to counting minterms, and operation (2) to cofactoring the encoding relation. $Count(f_i, g)$ can be computed as follows:

$$Count(f_i, g) = mint_z(\ \exists a[C(x, z) \wedge g(x)]\ ),$$

where $mint_z(h(z))$ is the number of minterms in function $h(z)$ depending on variables $z$.

Given a set of functions, computing the encoding relation is trivial; deriving codes from the encoding relation is not. The naïve implementation evaluates the following formula for each function in the set:

$$c_i(z) = \exists a[C(x, z) \wedge f_i(x)]$$

This leads to $\mu$ computations of the product with quantification, which is inefficient for large $\mu$.

## 5.3 Deriving Codes from Encoding Relation

A more efficient way of deriving codes requires only two partial traversals of the BDD of $C(x, z)$ but assumes that the code-bit variables $z$ are ordered *above* variables $x$. For clarity, BDDs without complement edges are considered in the sequel.

Each BDD node is annotated with the cofactors (*node->else* and *node->then*), the labeling variable (*node->var*), and the following additional data members: an integer counter (*node->count*), and the Boolean OR of the incoming BDD paths (*node->sum*). The exclamation mark (!) stands for complementation.

The goal of the first traversal (Figure 2) is to count the number of incoming edges of the nodes labeled with variables $z$ in the BDD of $C(x, z)$. This is achieved by associating a counter with each BDD node and incrementing the counter when the node is visited. Only the first visit to a node is followed by visiting the node's children. For this reason, each node is visited no more than once. The computational complexity of this traversal is linear in the number of BDD nodes labeled with variables $z$.

The second traversal (Figure 3) is more complicated. It involves computing the sum of the BDD paths converging into a node. Each visit to a node corresponds to a new path, which is added to the variable *node->sum* associated with the node (initially, it is set to the zero Boolean function for all nodes). Each time the node is visited, its counter of incoming edges, computed by the first traversal, is decremented. Upon the last traversal, the counter

becomes zero, meaning that we traversed all the paths leading to the given node and can now propagate the resulting sum of paths to the node's children.

The computation terminates at the nodes labeled by variables *x*. At this point, the computed sums of paths are equal to the codes of functions represented by the nodes.

```
CountEdges( bdd node )
{
    if ( node->count == 0 ) {
        if ( node->var ∈ code-bit variables ) {
            CountEdges( node->else );
            CountEdges( node->then );
        }
    }
    node->count = node->count + 1;
}
```

*Figure 2. Counting the number of incoming edges of the BDD nodes labeled by the code-bit variables.*

```
ComputePaths( bdd node, bdd paths )
{
    node->sum = node->sum ∨ paths;
    node->count = node->count - 1;
    if ( node->count == 0 ) {
        if ( node->var ∈ code-bit variables ) {
            ComputePaths( node->else,  node->sum ∧ !(node->var) );
            ComputePaths( node->then,  node->sum ∧  node->var );
        }
    }
}
```

*Figure 3. Computing OR of BDD paths leading to the BDD nodes labeled by the code-bit variables.*

## 5.4 Counting Overlapping Functions

Profiling of the encoding algorithm has shown that most of the runtime is spent in computing *Count*($f_i$, $g$). It is possible to speed-up this computation several times by developing a specialized BDD operator to count the number of minterms involving variables *z*, without evaluation of the formula for *Count*($f_i$, $g$), which includes the product and the existential quantification.

The pseudo-code of this operator is shown in Figure 4. It traverses BDDs of $C(x, z)$ and $g(z)$ without building new nodes and returns the number of minterms in the product $C(x, z) \wedge g(z)$. For this operator to work, variables *z* should be ordered *below* variables *x*. The procedure *CountMintSimple*(*C*) returns the number of minterms in *C* depending only on code-bit variables *z*.

```
CountMintSpecialized( bdd C, bdd g )
{
    v = TopMostVar( C, g );
    if ( v is a code-bit variable ) {
        assert( g = 0 || g = 1 );
        if ( g = 0 )        return 0;
        else                return CountMintSimple(C);
    }
    // otherwise,  v is a functional variable
    (C0, C1) = Cofactors( C, v );
    (g0, g1)  = Cofactors( g, v );
    return CountMintSpecialized(C0, g0) + CountMintSpecialized(C0, g0);
}
```

*Figure 4. A specialized BDD operator to count the number of overlapping functions.*

# 6 Application to LUT Cascade Synthesis

## 6.1 LUT Cascade

LUT cascade is a programmable device for evaluation of completely specified Boolean functions [11]. LUT cascade is approximately ten times faster than branching programs, even though it requires more memory. This observation gives LUT cascade a unique place among programmable devices and makes it a practical alternative to FPGAs.

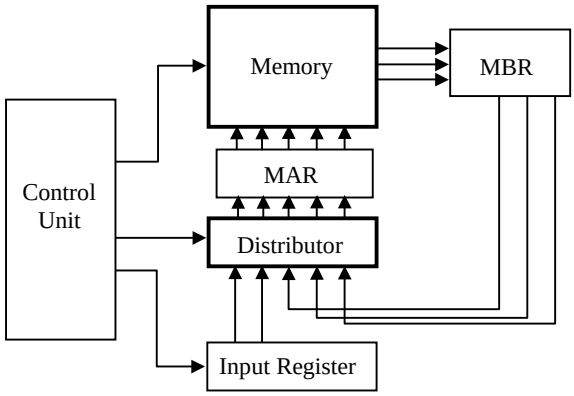


*Figure 5. Architecture of LUT cascade.*

LUT cascade consists of an array of look-up tables, denoted "Memory" in Figure 5, the control unit, the distributor, and several additional registers: the memory address register (MAR), the memory buffer register (MBR), and the input register to store the values of input variables. The distributor is an *n*-bit register storing the bit-vector applied to one stage of LUTs in each clocking period.

An *k*-input LUT can be programmed to implement any single-output Boolean function of *k* variables. A cascade of *k*-input LUTs can implement a single-output Boolean function if its BDD width ($\mu$), defined as the number of different cofactors on a level, is such that $\lceil \log_2 \mu \rceil < k$.

The evaluation of the function implemented in the LUT cascade is performed as follows. The input vector, for which the function is evaluated, is loaded into the input register. This vector is split into several parts loaded into the distributor at the successive clocking periods. Variables belonging to each part are determined during synthesis.

In the first clocking period, MAR is filled with the values of variables feeding into the first stage of the cascade. In the following clocking periods, one part of MAR is filled with the outputs of LUTs from the previous period; while the remaining part is filled with the input variables values. The evaluation continues until the last LUT is reached. This LUT produces the output value of the function.

*Example 4.* Consider function $F = (ab \vee c)d$. This function can be realized using a cascade of two-input LUTs. The structure of the cascade is found by mapping the BDD of *F* into three two-input LUTs. The K-maps of functions programmed in the LUTs are shown in Figure 6.

LUT-1 takes values of variables *a* and *b* and implements the function $x(a, b)$ distinguishing among the two cofactors in the

BDD on the level of variable *c*. The second stage of the cascade (LUT-2) depends on *x* and *c* and implements the function *y*(*x*, *c*) distinguishing among the two cofactors in the BDD on the level of variable *d*. Finally, the last LUT (LUT-3) takes *y* and *d* and produces the output value of *F*.

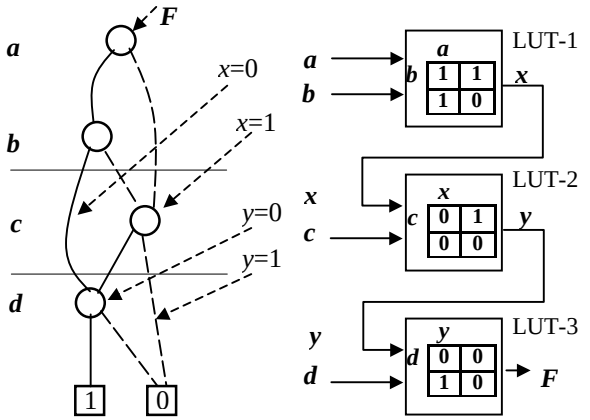


*Figure 6. LUT cascade for F(a,b,c,d) = (ab ∨ c)d.*

If the function has multiple outputs, they are encoded using additional variables. The resulting single-output function is called Encoded Characteristic Function for Non-zero outputs (ECFN) [11]. The evaluation of a particular output is performed by setting values of the additional variables to the code of this output. Using this method one output is evaluated at a time.

## 6.2 Encoding in LUT Cascade Synthesis

The theory and algorithms for LUT cascade synthesis are developed in [11]. Similar to the above example, LUT cascade synthesis is reduced to the decomposition of the BDD (or ECFN) of the function into a number of *n*-input blocks, each implemented by a LUT. Synthesizing one stage of the cascade involves encoding the cofactors found in the BDD below a certain level.

Different cofactor encodings lead to different cascade implementations. A reduction in the LUT count can be achieved by selecting decomposition subfunctions so that as many as possible are single-variable functions. The number of such subfunctions is equal to the number of LUTs replaced by a wire in the LUT cascade implementation.

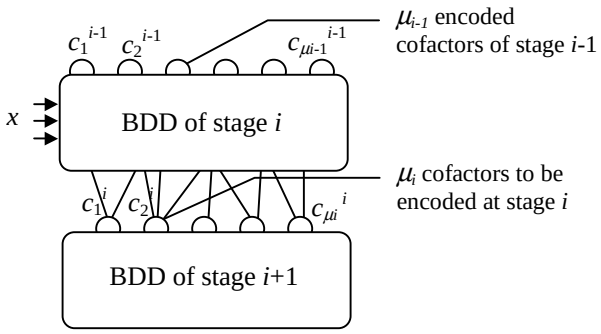


*Figure 7. Synthesis of one stage of LUT cascade.*

Encoding of cofactors is performed for each stage of the LUT cascade. Consider synthesis of stage *i* of the cascade in Figure 7. Suppose the number of cofactors coming from the previous stage *i*-1 is $\mu_{i-1}$. These cofactors can be encoded using $w_{i-1} = log_2 \mu_{i-1}$ bits. Then, *k*-input LUTs at stage *i* depend on $w_{i-1}$ outputs *z* of the previous stage and $k - w_{i-1}$ primary input variables *x*. Knowing the codes assigned to the cofactors at stage *i*-1, we label the nodes $c_1^{i-1}$, $c_2^{i-1}$,…, $c_{\mu i-1}^{i-1}$ with their codes and compute the sums of the BDD paths through stage *i* for nodes $c_1^i$, $c_2^i$,…, $c_{\mu k}^i$ the same way it was done in the procedure *ComputePaths*() in Figure 3.

The resulting path functions, $P_{\mu i}^i(x, z)$, depend on code-bit variables *z* at stage *i*-1 and the primary inputs *x* coming to stage *i*. The support size of $P_{\mu i}^i(x, z)$ does not exceed the number of LUT inputs. During the encoding, we perform a branch-and-bound procedure using variables in the support of $P_{\mu i}^i(x, z)$ and find the largest number of one-variable code-bit functions.

# 7 Experimental Results

The algorithm was programmed in C and included in EXTRA Library [7] extending the functionality of CUDD Release 2.3.1 [13]. The algorithm was tested on benchmarks used in [11].

The number of inputs in the LUTs was set to 15. For a function to be implementable using 15-input LUTs, each stage of the cascade should have at least one primary input variable, which makes the upper bound on the number of encoding bits equal to 14. The limit on the depth of branch-and-bound search was set to 5. This saved up to 5 LUTs in one stage due to encoding.

The following notation is accepted in Table 3. **SBDD** stands for the number of nodes in the shared BDD (w/complement edges) after reading and reordering. For some benchmarks, good variable orders [14] were used to build the BDDs for LUT cascade synthesis. **ECFN** is the number of nodes in the encoded characteristic function for non-zero outputs (w/o complement edges). **W_SBDD** is the maximum width of the shared BDD. **W_ECFN** is the maximum width of the ECFN.

**Stages** is the number of stages in the synthesized cascade. This number is equal to the number of 15-bit encoding problems solved. **Strict** is the number of LUTs using strict encoding. **Non-str** is the number of LUTs with non-strict encoding. **Read** is the time to read the benchmark from file. **Dec** is the time for decomposition without encoding. **Enc** is the time for encoding.

The last column of the table **N** lists the number of LUTs used in [11]. It should be noted that the experimental settings in [11] were different from ours. The latter work assumes a natural encoding of cofactors but considers the problem of encoding the outputs of the function in ECNF and the problem of finding a good variable order to reduce the BDD width. In our work, on the other hand, the main emphasis is on finding a non-strict encoding of cofactors, while the encoding of the outputs in ECNF was natural and variable reordering did not aim at reducing the BDD width.

The ECFNs and the LUT cascades derived by our program are written into the output BLIF files. Verification was not performed for “C7552.blif” because of the very large size of the output BLIF. For all other benchmarks, verification was successful.

In summary, columns **Strict** and **Non-str** show that the new encoding algorithm allows for saving up to 25% of LUTs. The runtime of the algorithm is comparable to the time needed for reading the benchmark. Taking into account the number of 14-bit encoding problems solved for each benchmark (column **Stages**), this runtime does not seem large.

## 8 Conclusions

This paper presents optimal non-strict encoding for sets of Boolean functions. An encoding is optimal if as many code-bit functions as possible are single-variable functions, or implement gates from a library. In LUT cascade synthesis, single-variable functions are implemented by a wire. As a result, the non-strict cofactor encoding reduces the numbers of LUTs in the LUT cascades by 25% on average, compared to the natural encoding.

The future work includes the application of the optimal non-strict encoding in the functional decomposition and technology-dependent decomposition-mapping of logic functions.

## References


[1] C. Files. *A new functional decomposition method as applied to machine learning and VLSI layout*. Ph. D. Thesis. Portland State University, June 2000.

[2] H. Gouji, T. Sasao, and M. Matsuura. On a method to reduce the number of LUTs in LUT cascades. *Technical Report of IEICE*, VLD2001-99, Nov. 2001 (in Japanese).

[3] J.-H. Jiang, Y. Jiang, R. K. Brayton. An implicit method for multi-valued network encoding. *Proc. IWLS'01*, pp.127-131.

[4] L. Lavagno, C. W. Moon, R. K. Brayton, A. L. Sangiovanni-Vincentelli. An efficient heuristic procedure for solving the state assignment problem for event-based specifications. *IEEE Trans. CAD*, 14-1, pp. 45-60, Jan 1995.

[5] Ch. Legl, B. Wurth, and K. Eckl. Computing support-minimal subfunctions during functional decomposition. *IEEE Trans. VLSI*, 6(3), pp. 354-363, Sept. 1998.

[6] G. D. Micheli, R. K. Brayton, A. L Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. *IEEE Trans. CAD*, 4-3, July 1985, pp. 269-285.

[7] A. Mishchenko. *EXTRA library of the DD procedures*. http://www.ee.pdx.edu/~alanmi/research/extra.htm

[8] R. Murgai, R. K. Brayton, A. Sangiovanni-Vincentelli. Optimum functional decomposition using encoding. *Proc. DAC '94*, pp. 408-414.

[9] V. Kravets, K. Sakallah. Constructive library-aware synthesis using symmetries. *Proc. DATE'00*, pp. 208-213.

[10] M. Perkowski et al. Decomposition of multiple-valued relations. *Proc. ISMVL '97*, Halifax, Canada, May 1997, pp.13-18.

[11] T. Sasao, M. Matsuura, Y. Iguchi. A cascade realization of multiple-output function for reconfigurable hardware, *Proc. IWLS '01*, pp. 225-230.

[12] T. Sasao. A new expansion of symmetric functions and their application to non-disjoint functional decompositions for LUT-type FPGAs. *Proc. IWLS*, pp. 105-110, June 2000.

[13] F. Somenzi. *CUDD package, Release 2.3.1*. http://vlsi.Colorado.EDU/~fabio/CUDD/cuddIntro.html

[14] F. Somenzi. Variable orders for MCNC benchmarks. ftp://vlsi.colorado.edu/pub/orders.tar.gz

[15] J. H. Tracey. Internal state assignment for asynchronous sequential machines. *IEEE Trans Elec. Comp.*, pp. 551-560. Aug. 1966.


*Table 3. Experimental results for LUT cascade synthesis with non-strict encoding.*

| Benchmark | | | Node Count | | Width | | LUT Cascade | | | Runtime, c | | | [11] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | Inputs | Outputs | SBDD | ECFN | W_SBDD | W_ECFN | Stages | Strict | Non-Str | Read | Dec | Enc | N |
| C432 | 36 | 7 | 1064 | 950 | 102 | 100 | 4 | 20 | 18 | 0.08 | 0.06 | 0.08 | 20 |
| C499 | 41 | 32 | 25866 | 27216 | 2112 | 2176 | 8 | 63 | 58 | 3.47 | 2.51 | 4.87 | 60 |
| C880 | 60 | 26 | 4053 | 4098 | 467 | 467 | 8 | 53 | 43 | 1.64 | 0.24 | 0.53 | 51 |
| C1908 | 33 | 25 | 5526 | 7449 | 595 | 620 | 5 | 36 | 33 | 1.04 | 0.57 | 1.61 | 35 |
| C2670 | 233 | 140 | 1850 | 2639 | 373 | 416 | 31 | 224 | 148 | 2.04 | 0.18 | 1.26 | 170 |
| C3540 | 50 | 22 | 23828 | 34712 | 4264 | 5414 | 17 | 194 | 144 | 16.45 | 3.08 | 28.01 | 107 |
| C5315 | 178 | 123 | 1719 | 2566 | 176 | 258 | 23 | 161 | 121 | 0.96 | 0.20 | 0.64 | 142 |
| C7552 | 207 | 108 | 2213 | 2939 | 177 | 193 | 26 | 176 | 147 | 6.17 | 0.20 | 0.86 | 156 |
| apex3 | 54 | 50 | 902 | 980 | 174 | 186 | 7 | 45 | 31 | 0.23 | 0.05 | 0.32 | 28 |
| apex7 | 49 | 37 | 304 | 346 | 101 | 87 | 6 | 29 | 20 | 0.01 | 0.01 | 0.08 | 21 |
| b9 | 16 | 5 | 78 | 93 | 30 | 30 | 2 | 5 | 5 | 0.00 | 0.00 | 0.00 | 14 |
| dalu | 75 | 16 | 689 | 581 | 147 | 147 | 10 | 64 | 41 | 0.41 | 0.04 | 0.39 | 40 |
| des | 256 | 245 | 2945 | 3024 | 622 | 363 | 36 | 268 | 172 | 1.26 | 0.36 | 2.13 | 235 |
| duke2 | 22 | 29 | 387 | 386 | 58 | 60 | 3 | 11 | 11 | 0.01 | 0.02 | 0.01 | 10 |
| e64 | 65 | 65 | 133 | 200 | 66 | 66 | 7 | 33 | 22 | 0.03 | 0.02 | 0.06 | 25 |
| ex4 | 128 | 28 | 509 | 534 | 71 | 49 | 12 | 47 | 41 | 0.02 | 0.01 | 0.05 | 32 |
| k2 | 45 | 45 | 1246 | 1299 | 274 | 262 | 6 | 36 | 31 | 0.16 | 0.05 | 0.19 | 28 |
| rot | 135 | 107 | 5922 | 7401 | 694 | 788 | 20 | 151 | 122 | 0.69 | 0.41 | 1.40 | 125 |
| spla | 16 | 46 | 637 | 616 | 99 | 95 | 2 | 7 | 6 | 0.30 | 0.03 | 0.02 | 8 |
| Total | | | | | | | 233 | 1623 | 1214 | 34.97 | 8.04 | 42.51 | 1307 |
| Ratio, % | | | | | | | | 100.0 | 74.8 | | | | 80.5 |