Eleventh International Workshop on Algebraic and Combinatorial Coding Theory
June 16-22, 2008, Pamporovo, Bulgaria *pp. 13-21*

Fast computing of the positive polarity Reed-Muller transform over $GF(2)$ and $GF(3)$

Valentin Bakoev v_bakoev@yahoo.com
University of Veliko Turnovo "St. Cyril and St. Methodius", BULGARIA

Krassimir Manev[1] manev@fmi.uni-sofia.bg
Faculty of Mathematics and Informatics, Sofia University, BULGARIA and
Institute of Mathematics and Informatics, Bulgarian Academy of Sciences,
8 G. Bonchev str., 1113 Sofia, BULGARIA

Abstract. The problem of efficient computing of binary and ternary positive (or zero) polarity Reed-Muller (PPRM) transform is important for many areas. The matrices, determining these transforms, are defined recursively or by Kronecker product. Using this fact, we apply the dynamic-programming strategy to develop three algorithms. The first of them is a new version of a previous our algorithm for performing the binary PPRM transform. The second one is a bit-wise implementation of the first algorithm. The third one performs the ternary PPRM transform. The last two algorithms have better time complexities in comparison with other algorithms, known to us.

1 Introduction

A well-known theorem in the theory of Boolean functions states that any Boolean function $f(x_{n-1}, x_{n-2}, \ldots, x_0)$ can be represented in an unique way by its *Zhegalkin polynomial*:

$$\begin{aligned} f(x_{n-1}, x_{n-2}, \ldots, x_0) &= a_0 \oplus a_1 x_0 \oplus a_2 x_1 \oplus a_3 x_1 x_0 \oplus \ldots \\ &\oplus \quad a_i x_{j_1} x_{j_2} \ldots x_{j_k} \oplus \cdots \oplus a_{2^n-1} x_{n-1} x_{n-2} \ldots x_0, \end{aligned} \tag{1}$$

where the coefficients $a_i \in \{0,1\}, 0 \leq i \leq 2^n - 1$, $i = 2^{j_1} + 2^{j_2} + \cdots + 2^{j_k}, j_1 > j_2 > \cdots > j_k$, and all variables are *positive* (uncomplemented). This canonical form is also known as *Positive Polarity Reed-Muller* (PPRM) expansion. If each variable x_i, $0 \leq i \leq n-1$, in (1) appears either uncomplemented, or complemented throughout, we obtain a *Fixed Polarity Reed-Muller* (FPRM) expansion. Let $p_i \in \{0,1\}$ denotes the polarity of x_i, $0 \leq i \leq n-1$, i.e. when $p_i = 0$ the polarity is positive (x_i is uncomplemented), and when $p_i = 1$ the

[1]This work was partially supported by the SF of Sofia University under Contract 171/05.2008.

polarity is negative (x_i is complemented). The function $f(x_{n-1}, x_{n-2}, \ldots, x_0)$ has a FPRM expansion of polarity $p, 0 \leq p \leq 2^n - 1$, when the integer p has a n-digit binary representation $p_{n-1}, p_{n-2} \ldots p_0$ and p_i is the polarity of x_i, for $i = n-1, n-2, \ldots, 0$. Thus f has 2^n FPRM possible expansions, each of them is a canonical form.

The FPRM binary transform is an important and known XOR-based expansion, having many applications in digital logic design, testability, fault detection, image compression, Boolean function decomposition, error correcting codes, classification of logic functions, and development of models for decision diagrams [2, 4, 5]. Because of the increasing interest in multiple-valued logic (MVL), the binary FPRM expansion has been extended to represent multiple-valued functions as well. Their FPRM expansions have also many applications in the just mentioned areas.

Every ternary function $f(x)$ of n-variable can also be represented by its canonical FPRM polynomial expansions as follows:

$$f_p(x_{n-1}, x_{n-2}, \ldots, x_0) = \sum_{i=0}^{3^n-1} a_i . \hat{x}_{n-1}^{k_{n-1}} \hat{x}_{n-2}^{k_{n-2}} \ldots \hat{x}_0^{k_0}, \tag{2}$$

where:

- all additions and multiplications are in $GF(3)$;
- i is the decimal equivalent of the n-digit ternary number $k_{n-1}k_{n-2} \ldots k_0$;
- $\hat{x}_j = x_j + p_j \in \{x_j, x_j + 1, x_j + 2\}$ is the literal of the j-th variable, in dependence of the polarity p_j. The required polarity is given (fixed) by the integer p, $0 \leq p \leq 3^n - 1$, which n-digit ternary representation is $p_{n-1}, p_{n-2} \ldots p_0$;
- the coefficient $a_i \in \{0, 1, 2\}$, $a_i = a_i(p)$ because it depends on the given polarity p;
- $\hat{x}_j^0 = 1$, $\hat{x}_j^1 = \hat{x}_j$ and $\hat{x}_j^2 = \hat{x}_j . \hat{x}_j$.

Optimization of FPRM transforms is an important problem in the area of logic design and spectral transforms. It concerns development of methods for determining the best FPRM representation of a given function among all possible FPRM expansions of it. The best is this one, which has minimal number of product terms or minimal number of literals. There are many approaches to perform such optimization.

Here we consider the problem: *"A Boolean (or ternary) function is given by its vector of functional values. Compute the vector of coefficients of its PPRM expansion"*. We represent three algorithms for fast solving of this problem. They can be used for computing of the rest FPRM expansions of a given function, as do this the algorithms and method in [4, 8]. The main idea of the proposed algorithms can be extended and applied for obtaining of other FPRM expansion, for computing PPRM expansions of MVL functions over other finite

fields, and also for fast computing of matrix-vector multiplication when the matrix is defined recursively (by Kronecker product).

2 Binary PPRM transform

Many scientists investigate the computing of binary FPRM transform – by applying of coefficient maps (Karnaugh maps folding, when the number of variables $n \leq 6$), coefficient matrix and tabular techniques [1, 6, 8, 10, 11]. All they consider algorithms for computing the PPRM transform in particular, and most of them apply a coefficient matrix approach. Let f be a n-variable boolean function, given by its vector of values $b = (b_0, b_1, \ldots, b_{2^n-1})$. The *forward* and *inverse* PPRM transform between the coefficient vector $a = (a_0, a_1, \ldots, a_{2^n-1})$ of Eq. (1) and the vector b is defined by the $2^n \times 2^n$ matrix M_n as follows [6, 9, 10]:

$$a^T = M_n.b^T, \quad \text{and} \quad b^T = M_n^{-1}.a^T \quad \text{over } GF(2). \tag{3}$$

The matrix M_n is defined recursively, as well as by Kronecker product:

$$M_1 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}, \quad M_n = \begin{pmatrix} M_{n-1} & O_{n-1} \\ M_{n-1} & M_{n-1} \end{pmatrix}, \text{ or } \quad M_n = M_1 \otimes M_{n-1} = \bigotimes_{i=1}^{n} M_1, \tag{4}$$

where M_{n-1} is the corresponding transform matrix of dimension $2^{n-1} \times 2^{n-1}$, and O_{n-1} is a $2^{n-1} \times 2^{n-1}$ zero matrix. Furthermore $M_n = M_n^{-1}$ over $GF(2)$, and hence the forward and the inverse transform are performed in an uniform way. So we shall consider only the forward one. In all papers known to us, there are not complete description of the algorithm for computing of such transform, defined by equalities (3) and (4). These equalities are derived in [6] (Theorem 2) and computing of the transform is illustrated by an example, almost the same is done in [9]. In [1] some equalities, which concern computing of the coefficients of the vector a and relations between them, are derived. Computing of the PPRM transform in [10] is illustrated by its "butterfly" (or "signal flow") diagram only.

Ten years ago we have proposed an algorithm for fast computing the PPRM transform (called by as "Zhegalkin transform") [7]. We developed this algorithm independently of other authors, because their papers in this area were unknown (unaccessible) to us at this time. Here we propose another version of this algorithm, created by the dynamic-programming approach. We also comment its bit-wise implementation, which improves significantly the previous time and space complexity. The same approach will be applied for fast computing of the PPRM transform over $GF(3)$.

Let v be a vector, $v \in \{0,1\}^{2^n}$. We could consider each position of the vector v *labeled* with the corresponding vector of $\{0,1\}^n$, so that the labels are

ordered lexicographically. Let $\alpha \in \{0,1\}^k$, $1 \leq k < n$. We will denote by $v_{[\alpha]}$ the sub-vector of these positions in v, first k-coordinates of labels of which are fixed to α. We can rewrite Eq. (3) as follows:

$$a^T = M_n.b^T = \begin{pmatrix} M_{n-1} & O_{n-1} \\ M_{n-1} & M_{n-1} \end{pmatrix} \begin{pmatrix} b^T_{[0]} \\ b^T_{[1]} \end{pmatrix} \tag{5}$$
$$= \begin{pmatrix} M_{n-1}.b^T_{[0]} \\ M_{n-1}.b^T_{[0]} \oplus M_{n-1}.b^T_{[1]} \end{pmatrix} = \begin{pmatrix} a^T_{[0]} \\ a^T_{[1]} \end{pmatrix}.$$

Therefore:

$$\begin{aligned} a^T_{[0]} &= M_{n-1}.b^T_{[0]}, \\ a^T_{[1]} &= M_{n-1}.b^T_{[0]} \oplus M_{n-1}.b^T_{[1]} = a^T_{[0]} \oplus M_{n-1}.b^T_{[1]}. \end{aligned} \tag{6}$$

The last two equalities *define recursively* the solution of the problem. They demonstrate how it can be constructed by the solutions of its subproblems. So the problem exhibits the *optimal substructure property* – the first key ingredient for applying the dynamic-programming strategy. The second one – *overlapping subproblems* – is also shown in (6). If we are computing a recursively, we have to compute first $a_{[0]}$ (recursively). Then we have to compute $a_{[1]}$ (recursively) and this will imply computing of $a_{[0]}$ again.

To apply the dynamic-programming strategy we will replace the recursion by an iteration and will compute the vector a *"bottom-up"*. The main idea can be drawn from last two equalities – if we make one more step, expressing M_{n-1} by M_{n-2} and replacing $a_{[0]}$ by $(a_{[00]}, a_{[01]})$, $a_{[1]}$ by $(a_{[10]}, a_{[11]})$, $b_{[0]}$ by $(b_{[00]}, b_{[01]})$, $b_{[1]}$ by $(b_{[10]}, b_{[11]})$, and so on. We conclude that the iteration should perform n steps. Starting from the vector b (as an input), at k-th step, $k = 1, 2, \ldots, n$, we consider the current vector b as divided into two kinds of blocks: *source* and *target*, which alternate with each other. All they have a *size*, equal to 2^{k-1}. At each step, every source block is *added* (by a component-wise XOR) to the next block, which is its target block. The result is assigned to the current vector b. So, after these n steps, the vector b is transformed to the vector a. Assuming that the vector b is represented by an array `b` of 2^n bytes, the pseudo code of this algorithm is:

```
Binary_PPRM (b, n)
1)  blocksize = 1;
2)  for k = 1 to n do 3) source = 0;   //start of the source block
4)     while source < 2^n do
5)        target = source + blocksize; //start of the target block
6)        for i = 0 to blocksize - 1 do
             //component-wise XOR over current blocks
7)           b[target + i] = b[target + i] XOR b[source + i];
```

```
          //start of the next source block
8)        source = source + 2 * blocksize;
9)     blocksize = 2 * blocksize;
10) return b;                          //b is transformed to a
```

The correctness of the algorithm can be proved easily by induction on n. In its k-th step, $1 \leq k \leq n$, there are 2^{n-k} source blocks and so many target blocks, each of size 2^{k-1}. The algorithm adds (XORs) these source blocks to the corresponding target blocks, and so it performs $2^{k-1}.2^{n-k} = 2^{n-1}$ XORs in the k-th step. Therefore, when the input size is 2^n, the algorithm has a time complexity $\Theta(n.2^{n-1})$ and $\Theta(2^n)$ space complexity. They are many times better than the corresponding complexities, which we shall obtain if we generate the matrix M_n and compute directly the matrix-vector multiplication, given by Eq. (3).

Now we discuss a new version of the given algorithm, obtained by applying a bit-wise representation of the vector b and bit-wise operations. Let $d = 2^j$ be the size (in bits) of the computer word. Then $m = \lceil 2^{n-j} \rceil$ computer words are sufficient to represent the vector b. For simplicity, let $n = j$ (i.e. $m = 1$), and we denote by B the representation of b as a binary number. We use an additional integer *temp*, initialized by $temp = B$. In *temp* we set the values in the target blocks to zero – i.e. we mask them by zeros, and the values in the source blocks we remain the same – we mask them by ones. For that purpose, in the k-th step ($k = 1, 2, \ldots n$) we should use a mask: `mask[k]`$=\underbrace{11\ldots1}_{2^{k-1}}\underbrace{00\ldots0}_{2^{k-1}}\cdots\underbrace{11\ldots1}_{2^{k-1}}\underbrace{00\ldots0}_{2^{k-1}}$, where 2^{k-1} is the block size. After that, we "shift right" them by 2^{k-1} positions and so the source blocks are moved to the places of the target blocks, corresponding to them. Finally, we compute a bit-wise XOR between B and *temp* and store the result in B. So, the body of the main cycle in row 2 of the given above pseudo code (i.e. the rows 3, 4,..., 9) could be replaced by:

```
3)   temp = B AND mask[k];         //masks the blocks;
4)   temp = temp SHR blocksize;    //shift right
5)   B = B XOR temp;               //XOR between all blocks.
6)   blocksize = blocksize SHL 1; //double the blocksize
```

We have only four bit-wise operations, repeated n times. Therefore the time complexity of this version of the algorithm is $\Theta(n)$. The array `mask` consists of n computer words and they can be pre-computed once in $\Theta(n^2)$ time and not considered as a part of algorithm. When $n > j$ then $m = 2^{n-j} > 1$ words of memory will be necessary. In this case, during the steps $1, 2, \ldots, j$, the instructions 3, 4 and 5 of the algorithm will be executed for each word

separately. During the steps $j+1, j+2, \ldots, n$ the masks are no more necessary, because the blocks are composed of whole words. In such way only $m/2$ XOR operations will be necessary on each of these steps. Finally, the time complexity becomes $\Theta(m.n)$ generally, which is the best one, known to us.

To compare these two versions we have generated all Boolean functions of 5 variables and perform the PPRM transform over each of them. When the new version of the algorithm uses a 32-bit computer word and generates the masks only once, it runs 22 times faster.

3 Ternary PPRM transform

The ternary FPRM and some other transforms are investigated intensively by Falkowski, Fu, etc. [2, 3, 4, 5]. These transforms are determined by the corresponding matrices, defined recursively or by Kronecker product. These matrices are used for building *recursive* algorithms, performing these expansions. Computing of the ternary PPRM transform is an important part for some of them or for other fast algorithms [4]. Let $f(x_{n-1}, x_{n-2}, \ldots, x_0)$ be a ternary function, represented by its vector of values $b = (b_0, b_1, \ldots, b_{3^n-1})$. Analogously to the binary case, the ternary *forward* PPRM transform between the coefficient vector $a = (a_0, a_1, \ldots, a_{3^n-1})$ and the vector b is defined by the $3^n \times 3^n$ matrix T_n as follows [2, 3, 4]:

$$a^T = T_n.b^T \text{ over } GF(3). \tag{7}$$

The matrix T_n is defined recursively, or by Kronecker product:

$$T_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 1 \\ 2 & 2 & 2 \end{pmatrix}, \quad T_n = \begin{pmatrix} T_{n-1} & O_{n-1} & O_{n-1} \\ O_{n-1} & 2.T_{n-1} & T_{n-1} \\ 2.T_{n-1} & 2.T_{n-1} & 2.T_{n-1} \end{pmatrix}, \text{or} \quad T_n = T_1 \otimes T_{n-1} = \bigotimes_{i=1}^{n} T_1, \tag{8}$$

where T_{n-1} is the corresponding transform matrix of dimension $3^{n-1} \times 3^{n-1}$, and O_{n-1} is a $3^{n-1} \times 3^{n-1}$ zero matrix. It is easy to see that $T_n \neq T_n^{-1}$, and so the forward and inverse ternary PPRM transforms do not coincide.

Let v be a vector, $v \in \{0,1,2\}^{3^n}$. We consider each position of v *labeled* with the corresponding vector of $\{0,1,2\}^n$, so that the labels are ordered lexicographically. Let the vector $\alpha \in \{0,1,2\}^k$ and $1 \leq k < n$. We denote by $v_{[\alpha]}$ the sub-vector of these positions in v, first k-coordinates of labels of which are fixed to α. Using Eq. (8), we rewrite Eq. (7) as:

$$\begin{aligned} a^T = T_n.b^T &= \begin{pmatrix} T_{n-1} & O_{n-1} & O_{n-1} \\ O_{n-1} & 2.T_{n-1} & T_{n-1} \\ 2.T_{n-1} & 2.T_{n-1} & 2.T_{n-1} \end{pmatrix} \begin{pmatrix} b_{[0]}^T \\ b_{[1]}^T \\ b_{[2]}^T \end{pmatrix} = \\ &= \begin{pmatrix} T_{n-1}.b_{[0]}^T & & \\ & 2.T_{n-1}.b_{[1]}^T & + \; T_{n-1}.b_{[2]}^T \\ 2.T_{n-1}.b_{[0]}^T & +2.T_{n-1}.b_{[1]}^T & +2.T_{n-1}.b_{[2]}^T \end{pmatrix} = \begin{pmatrix} a_{[0]}^T \\ a_{[1]}^T \\ a_{[2]}^T \end{pmatrix} \text{ over } GF(3), \end{aligned} \tag{9}$$

Therefore:

$$\begin{aligned} a^T_{[0]} &= T_{n-1}.b^T_{[0]} \\ a^T_{[1]} &= 2.T_{n-1}.b^T_{[1]} + T_{n-1}.b^T_{[2]} \\ a^T_{[2]} &= 2.(T_{n-1}.b^T_{[0]} + T_{n-1}.b^T_{[1]} + T_{n-1}.b^T_{[2]}) \end{aligned} \tag{10}$$

The last equalities determine the solution recursively. The reasons to apply the dynamic-programming strategy are the same as in the binary case. The final solution can be obtained by the solutions of its subproblems (i.e. when the matrix-vector multiplications of the type $T_{n-1}.b^T_{[i]}$ are already computed) by 3 additions of vectors and 2 multiplications of vector by a scalar in $GF(3)$. Thinking about them as source and target blocks, we shall replace them by 4 additions between blocks in $GF(3)$, as it is shown in Fig. 1, for $n = 1$. Obviously, some source and target blocks (of size 1, when $n = 1$) change their roles.

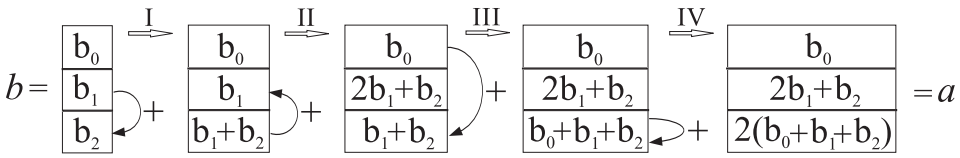

Figure 1: For $n = 1$, vector b is transformed to vector a by 4 additions in $GF(3)$.

The same model of computing will be valid if we expand the equalities (9) and (10) completely for $n = 2$. In the first step we apply the scheme of additions in Fig. 1 for each of sub-vectors $b_{[0]}, b_{[1]}$ and $b_{[2]}$. In the second step we consider the resulting sub-vectors as blocks of size 3, labeled by $0, 1$, and 2, respectively. We compute component-wise additions between the blocks, following the scheme in Fig. 1 and so we obtain the vector a.

We can extend this model of computing for an arbitrary n. Thus we obtain an algorithm, which starts from the given vector b (as an input) and performs n steps. At each step, the current vector b (as a result of a previous step) is divided into blocks of size 3^{k-1}, where k is the number of the step. The blocks are labeled by $0, 1, \ldots, 3^{n-k+1}$. For each triple of consecutive blocks the algorithm performs component-wise additions (in $GF(3)$) between the blocks in the triple, following the scheme in Fig. 1. So, before the last step, the sub-vectors (blocks) $T_{n-1}.b^T_{[i]}$, labeled by $i = 0, 1, 2$, are already computed. In the last step, the algorithm performs the additions between the blocks in the last triple, as they are given in Fig. 1, and so it obtains the vector a. If the vector b is represented by an array `b` of 3^n bytes, the pseudo code of this algorithm is:

```
Ternary_PPRM (b, n)
1)  blocksize = 1;
2)  for k = 1 to n do
3)     base = 0;   //start of the 0-blocks in a current triple
4)     while base < 3^n do
5)        first = base + blocksize;           //start of 1-block
6)        second = first + blocksize;         //start of 2-block
7)        AddBlock(first,second,blocksize );  //adds 1-bl. to 2-bl.
8)        AddBlock(second,first,blocksize );  //adds 2-bl. to 1-bl.
9)        AddBlock(base,second,blocksize );   //adds 0-bl. to 2-bl.
10)       AddBlock(second,second,blocksize ); //adds 2-bl. to itself
11)       base = base + 3*blocksize;          //start next triple
12)    blocksize= 3*blocksize;
13) return b;      //b is transformed to a
```

Procedure `AddBlock (s, t, size)` adds the block (sub-vector), starting from coordinate `s`, to the block, starting from coordinate `t`. It performs `size` component-wise additions by a table look-up (of additions in $GF(3)$), since this is faster than modular arithmetic.

The arguments above the pseudo code and equalities (9) and (10) imply the correctness of the algorithm. Following them, it can be proved strongly by induction on n. The space complexity of the algorithm is $\Theta(3^n)$, the same as the size of input. Its time complexity is derived easily. In the k-th step, $1 \leq k \leq n$, the size of the blocks is 3^{k-1}, and for each triple of blocks the algorithm performs 4.3^{k-1} additions. There are $3^n/(3.3^{k-1}) = 3^{n-k}$ triples, and so the additions in the k-th step are $4.3^{k-1}.3^{n-k} = 4.3^{n-1}$. Therefore the time complexity is $\Theta(n.3^{n-1})$. For comparison, in [4] the authors refer to an algorithm for fast computing of ternary PPRM transform, which performs $n.3^n$ additions and $4n.3^{n-1}$ multiplications.

The matrix T_n^{-1} can be expressed by equalities analogous to these in (8), hence the inverse transform can be performed in way, similar to the performing of forward transform.

4 Conclusions

Here we have used the dynamic-programming strategy to develop three algorithms. They are based on matrices, defined recursively or by Kronecker product, which determine the PPRM transforms over $GF(2)$ and $GF(3)$. The model of building the given algorithms can be extended and applied for fast computing of other FPRM expansions over the considered fields, for other finite fields with prime number of elements, or for fast computing of matrix-vector multiplication

when the matrix is defined recursively. Proposed algorithms have better time complexities in comparison with other algorithms, known to us.

References

[1] A. Almaini, P. Thomson, D. Hanson, Tabular techniques for Reed-Muller logic, *Int. J. Electronics* 70, 1991, 23-34.

[2] B. Falkowski, C. Fu, Fastest classes of linearly independent transforms over GF(3) and their properties, *IEE Proc. Comput. Digit. Tech.* 152, 2005, 567-576.

[3] B. Falkowski, C. Fu, Polynomial expansions over GF(3) based on fastest transformation, *Proc. 33-rd Intern. Symp. Mult.-Val. Logic*, 2003, 40-45.

[4] B. Falkowski, C. Lozano, Column polarity matrix algorithm for ternary fixed polarity Reed-Muller expansions, *J. Circ., Syst., Comp.* 15, 2006, 243-262.

[5] C. Fu, B. Falkowski, Ternary fixed polarity linear Kronecker transforms and their comparison with ternary Reed Muller transform, *J. Circ., Syst., Comp.* 14, 2005, 721-733.

[6] B. Harking, Efficient algorithm for canonical Reed-Muller expansions of Boolean functions, *IEE Proc. Comput. Digit. Tech.* 137, 1990, 366-370.

[7] K. Manev, V. Bakoev, Algorithms for performing the Zhegalkin tranformation, *Proc. XXVII Spring Conf. UBM*, 1998, 229-233.

[8] M. Perkowski, L. Jozwiak, R. Drechsler, A canonical AND/EXOR form that includes both the generalized Reed-Muller forms and Kronecker Reed-Muller forms, *Proc. RM'97*, Oxford Univ., 1997, 219-233.

[9] P. Porwik, Efficient calculation of the Reed-Muller form by means of the Walsh transform, *Int. J. Appl. Math. Comput. Sci.* 12, 2002, 571-579.

[10] S. Rahardja, B. Falkowski, C. Lozano, Fastest linearly independent transforms over GF(2) and their properties, *IEEE Trans. Circuts Syst.* 52, 2005, 1832-1844.

[11] E. Tan, H. Yang, Fast tabular technique for fixed-polarity Reed-Muller logic with inherent parallel processes, *Int. J. Electr.* 85, 1998, 511-520.