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

Valentin Bakoev

(University of V. Turnovo ”St. Cyril and St. Methodius”, Bulgaria)

and

Krassimir Manev

(University of Sofia ”St. Kliment Ohridski”, Bulgaria)

1. Introduction

The known theorem of Zhegalkin 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*:

$$(1) \qquad f(x_{n-1}, x_{n-2}, \ldots, x_0) = a_0 \oplus a_1x_0 \oplus a_2x_1 \oplus \\ \oplus a_3x_1x_0 \oplus \ldots \oplus a_ix_{j_1}x_{j_2} \ldots x_{j_k} \oplus \ldots \\ \oplus a_{2^n-1}x_{n-1}x_{n-2} \ldots x_0,$$

where the coefficients $a_i \in \{0, 1\}, 0 \leq i \leq 2^n - 1$.
The n-digit binary representation of i is a characteristic vector of the variables in the corresponding product, and all variables are *positive* (uncomplemented).

1. Introduction

This canonical form is also known as
Positive Polarity Reed-Muller (PPRM) expansion.

When 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.

1. Introduction

Let $p_i \in \{0, 1\}$ denotes the polarity of x_i:
if $p_i = 0$ the polarity is *positive* (x_i is uncomplemented), and
if $p_i = 1$ the polarity is *negative* (x_i is complemented),
for $0 \le i \le n-1$.

The function $f(x_{n-1}, x_{n-2}, \ldots, x_0)$ has a FPRM expansion of polarity p, $0 \le p \le 2^n - 1$, when the integer $p = p_{n-1}p_{n-2}\ldots p_{0(2)}$ and p_i is the polarity of x_i, for $i = n-1, n-2, \ldots, 0$.

1. Introduction

The binary FPRM transform has many applications in:

- digital logic design;
- testability;
- fault detection;
- image compression;
- error correcting codes;
- Boolean function decomposition;
- classification of logic functions;
- development of models for decision diagrams, etc.

1. Introduction

Because of the increasing interest in multiple-valued logic (MVL), the binary FPRM expansion has been extended to represent multiple-valued functions. Their FPRM expansions have also many applications in the just mentioned areas.

Every ternary function f of n-variables can also be represented by its canonical FPRM polynomial expansions as:

1. Introduction

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

- all additions and multiplications are in $GF(3)$;
- i is the decimal equivalent of $k_{n-1}k_{n-2}\ldots k_{0(3)}$;
- $\hat{x}_j = x_j + p_j \in \{x_j, x_j + 1, x_j + 2\}$ is the literal of the j-th variable, in dependence of the polarity p_j;
- the polarity is given by the integer p, $0 \leq p \leq 3^n - 1$, so that $p = p_{n-1}p_{n-2}\ldots p_{0(3)}$;
- the coefficient $a_i \in \{0, 1, 2\}$, $a_i = a_i(p)$;
- $\hat{x}_j^0 = 1$, $\hat{x}_j^1 = \hat{x}_j$ and $\hat{x}_j^2 = \hat{x}_j . \hat{x}_j$.

1. Introduction

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.

1. Introduction

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.

2. Binary PPRM transform

The computing of binary FPRM transform is investigated by many scientists:

- Wu, Tran etc. use coefficient maps (Karnaugh maps folding, when $n \leq 6$);
- Almaini, Tan, Yang etc. apply tabular techniques;
- Green, Harking, Porwik, Perkowski, Falkowski etc. use coefficient matrices.

All they consider algorithms for computing of the PPRM transform in particular.

2. Binary PPRM transform

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:

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

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

2. Binary PPRM transform

(4) $$M_1 = \begin{pmatrix} 1\,0 \\ 1\,1 \end{pmatrix}, \quad M_n = \begin{pmatrix} M_{n-1}\; O_{n-1} \\ M_{n-1}\; M_{n-1} \end{pmatrix},$$

$$or \quad M_n = M_1 \otimes M_{n-1} = \bigotimes_{i=1}^{n} M_1,$$

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. $M_n = M_n^{-1}$ over $GF(2)$, so the forward and the inverse transform are performed in an uniform way.

2. Binary PPRM transform

In all papers known to us, there is not complete description of an algorithm for computing of PPRM transform, defined by Eq. (3) and (4). These equalities are derived by Harking and computing of the transform is illustrated by an example, almost the same is done by Porwik. Almaini etc. derive some equalities, which concern computing of the coefficients (i.e. coordinates of the vector a) and relations between them.
Most of authors illustrate the computing of the PPRM transform by its "butterfly" (or "signal flow") diagrams only.

2. Binary PPRM transform

Ten years ago we have proposed an algorithm for fast computing of the PPRM transform (called by as "Zhegalkin transform"). 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 $GF(3)$.

2. Binary PPRM transform

Using Eq. (4), we can rewrite Eq. (3) as:

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

where $v_{[0]}$ (resp. $v_{[1]}$) denotes the sub-vector of the n-dimensional binary vector v, which coordinates are labeled by n-digit binary numbers, beginning with 0 (resp. 1).

2. Binary PPRM transform

Therefore:

$$
\begin{aligned}
a_{[0]}^T &= M_{n-1}.b_{[0]}^T, \\
a_{[1]}^T &= M_{n-1}.b_{[0]}^T \oplus M_{n-1}.b_{[1]}^T = a_{[0]}^T \oplus M_{n-1}.b_{[1]}^T.
\end{aligned}
\tag{6}
$$

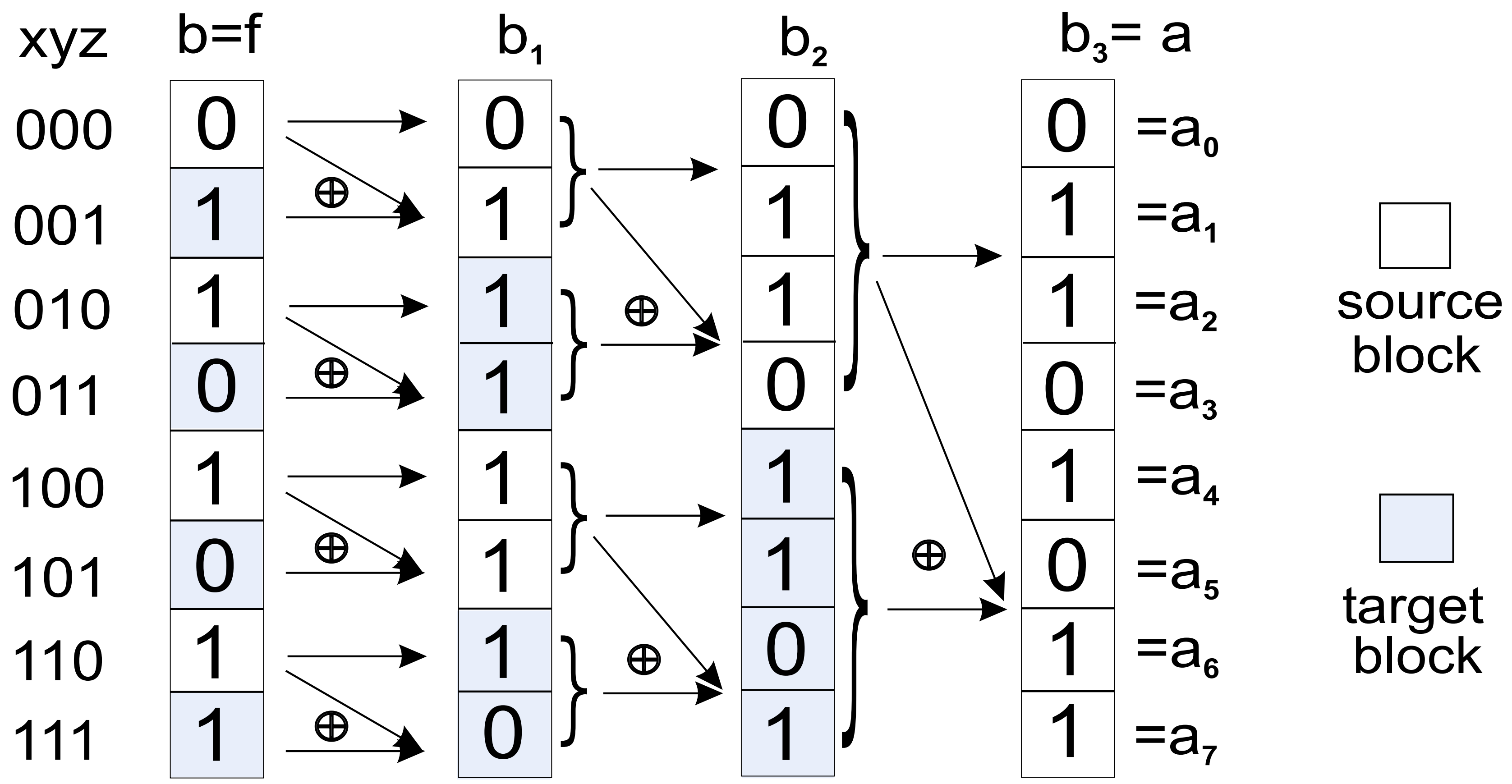
These equalities *define recursively* the solution of the problem. They show how it can be obtained by the solutions of its subproblems. So the problem exhibits the *optimal substructure property* – the first key reason to apply the dynamic-programming strategy. The second one – *overlapping subproblems* – is also shown in (6). If we are computing a recursively, we have to compute first $a_{[0]}$ (recursively), and afterward $a_{[1]}$ (recursively), which implies computing of $a_{[0]}$ again.

2. Binary PPRM transform

We will replace the recursion by an iteration and will compute the vector a *"bottom-up"*. The iteration should perform n steps. Starting from the vector b (as an input), at k-th step, $k = 1, 2, \dots, n$, we consider the current vector b as divided into two kinds of blocks: *source* and *target*, of $size = 2^{k-1}$, which alternate with each other. 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. After n steps, the vector b is transformed to vector a.

2. Binary PPRM transform

Example. Let $f(x, y, z) = (0, 1, 1, 0, 1, 0, 1, 1) = b$.



Therefore $f(x, y, z) = z \oplus y \oplus x \oplus xy \oplus xyz$.

2. Binary PPRM transform

If the vector b is represented by an array b of 2^n bytes, the pseudo code of this algorithm is:

```
Binary_PPRM_Transform (b, n)
1)  blocksize= 1;
2)  for k= 1 to n do
3)     source= 0;   //beginning of the source block
4)     while source < 2^n do
5)        target= source + blocksize; //beg. of target bl.
6)        for i= 0 to blocksize-1 do
7)           b[target+i]= b[target+i] XOR b[source+i];
8)        source= source+2*blocksize; //beg. of source bl.
9)     blocksize= 2*blocksize;
10) return b;   //b is already transformed to a
```

2. Binary PPRM transform

The correctness of the algorithm can be proved easily by induction on n.

When the input size is 2^n, the algorithm has a time complexity $\Theta(n.2^{n-1})$ and space complexity $\Theta(2^n)$.

2. Binary PPRM transform

The new version of this algorithm is obtained by applying a bit-wise representation of the vector b and bit-wise operations: masks, shifts, XORs. When the bit-wise representation of vector b takes up m computer words, the time complexity becomes $\Theta(m.n)$ generally. This is the best one, known to us.

For comparison, we have generated all Boolean functions of 5 variables and we have performed the PPRM transform over each of them. When the new version of the algorithm uses a 32-bit computer word, it runs 22 times faster.

3. Ternary PPRM transform

The ternary FPRM and some other transforms are investigated intensively by Falkowski, Fu, Lozano, etc. 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.

3. Ternary PPRM transform

Let $f(x_{n-1}, x_{n-2}, \dots, x_0)$ be a ternary function, represented by its vector of values $b = (b_0, b_1, \dots, b_{3^n-1})$. It is known that the ternary *forward* PPRM transform between the coefficient vector $a = (a_0, a_1, \dots, a_{3^n-1})$ in Eq. (2) and the vector b is defined by the $3^n \times 3^n$ matrix T_n as:

(7) $$a^T = T_n.b^T \;\; over \;\; GF(3).$$

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

3. Ternary PPRM transform

(8)
$$T_1 = \begin{pmatrix} 1\,0\,0 \\ 0\,2\,1 \\ 2\,2\,2 \end{pmatrix}, \qquad T_n = \begin{pmatrix} T_{n-1} & O_{n-1} & O_{n-1} \\ O_{n-1} & 2.T_{n-1} & T_{n-1} \\ 2.T_{n-1} & 2.T_{n-1} & 2.T_{n-1} \end{pmatrix}, \ or$$

$$T_n = T_1 \otimes T_{n-1} = \bigotimes_{i=1}^{n} T_1,$$

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.

3. Ternary PPRM transform

Using Eq. (8), we rewrite Eq. (7) as:

$$a^T = T_n.b^T = \begin{pmatrix} T_{n-1} & O_{n-1} & O_{n-1} \\ O_{n-1} & 2.T_{n-1} & T_{n-1} \\ 2.T_{n-1} & 2.T_{n-1} & 2.T_{n-1} \end{pmatrix} \begin{pmatrix} b^T_{[0]} \\ b^T_{[1]} \\ b^T_{[2]} \end{pmatrix} = \tag{9}$$

$$= \begin{pmatrix} T_{n-1}.b^T_{[0]} & & \\ & 2.T_{n-1}.b^T_{[1]} & + \ T_{n-1}.b^T_{[2]} \\ 2.T_{n-1}.b^T_{[0]} & +2.T_{n-1}.b^T_{[1]} & +2.T_{n-1}.b^T_{[2]} \end{pmatrix} = \begin{pmatrix} a^T_{[0]} \\ a^T_{[1]} \\ a^T_{[2]} \end{pmatrix} \ \ over \ \ GF(3),$$

where $v_{[0]}$ (resp. $v_{[1]}, v_{[2]}$) denotes the sub-vector of the n-dimensional ternary vector v, which coordinates are labeled by n-digit ternary numbers, beginning with 0 (resp. $1, 2$).

3. Ternary PPRM transform

Therefore:

$$(10)\quad \begin{aligned} a_{[0]}^T &= T_{n-1}.b_{[0]}^T \\ a_{[1]}^T &= 2.T_{n-1}.b_{[1]}^T + T_{n-1}.b_{[2]}^T \\ a_{[2]}^T &= 2.(T_{n-1}.b_{[0]}^T + T_{n-1}.b_{[1]}^T + T_{n-1}.b_{[2]}^T) \end{aligned}$$

Equalities (10) determine the solution recursively. The reasons to apply the dynamic-programming strategy are the same as in the binary case. If the multiplications of the type $T_{n-1}.b_{[i]}^T$ are already computed, the final solution can be obtained by 3 additions of vectors and 2 multiplications of vector by a scalar in $GF(3)$.

3. Ternary PPRM transform

Thinking about them as source and target blocks, we replace them by 4 additions between blocks in $GF(3)$, as it is shown on Fig. 1, for $n = 1$. Obviously, some source and target blocks (of size 1, when $n = 1$) change their roles.

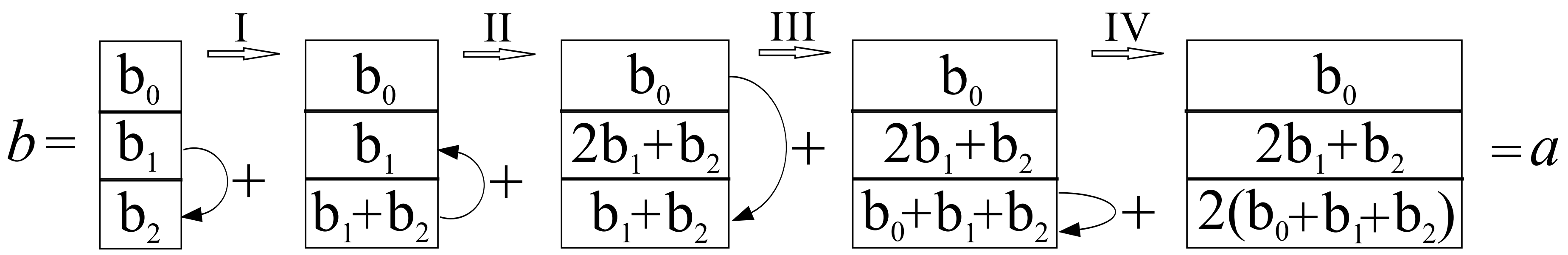


Figure 1: For $n = 1$, vector b is transformed to vector a by 4 additions in $GF(3)$.

3. Ternary PPRM transform

We extend this model of computing for an arbitrary n. So we obtain an algorithm, which starts from the given vector b (as an input) and performs n steps. At the k-th step, the current vector b is divided into 3^{n-k+1} blocks of size 3^{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.

If the vector b is represented by an array b of 3^n bytes, the pseudo code of this algorithm is:

3. Ternary PPRM transform

```
Ternary_PPRM_Transform (b, n)
1)  blocksize= 1;
2)  for k= 1 to n do
3)     base= 0;   //beg. of the 0-block
4)     while base < 3^n do
5)        first= base + blocksize;   //beg. of I block
6)        second= first + blocksize; //beg. of II block
7)        AddBlock ( first, second, blocksize );
8)        AddBlock ( second, first, blocksize );
9)        AddBlock ( base, second, blocksize );
10)       AddBlock ( second, second, blocksize );
11)       base= base + 3*blocksize;  //beg. next triple
12)    blocksize= 3*blocksize;
13) return b;   //b is already transformed to a
```

3. Ternary PPRM transform

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 in $GF(3)$ between these blocks.

The correctness of the algorithm 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 $\Theta(n.3^{n-1})$.

3. Ternary PPRM transform

For comparison, Falkowski and Lozano[a] refer to an algorithm for fast computing of ternary PPRM transform, which performs $n.3^n$ additions and $4n.3^{n-1}$ multiplications.

[a]Column polarity matrix algorithm for ternary fixed polarity Reed–Muller expansions”, *J. of Circuits, Systems, and Computers*, Vol. 15, No. 2, 2006, pp. 243-262)

4. Conclusions

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, or other finite fields with prime number of elements;
- matrix-vector multiplication when the matrix is defined recursively.

The proposed algorithms have better time complexities in comparison with other algorithms, known to us.