An Efficient Technique for Synthesis and Optimization of Polynomials in GF(2^m)

Abusaleh M. Jabir
School of Technology
Oxford Brookes University
Oxford OX3 0BP, UK
ajabir@brookes.ac.uk

Dhiraj K. Pradhan[*]
Department of Computer Science
University of Bristol
Bristol BS8 1UB, UK
pradhan@cs.bris.ac.uk

Jimson Mathew
Department of Computer Science
University of Bristol
Bristol BS8 1UB, UK
jimson@cs.bris.ac.uk

ABSTRACT

This paper presents an efficient technique for synthesis and optimization of polynomials over GF(2^m), where m is a non-zero positive integer. The technique is based on a graph-based decomposition and factorization of polynomials over GF(2^m), followed by efficient network factorization and optimization. A technique for efficiently computing coefficients over GF(p^m), where p is a prime number, is first presented. The coefficients are stored as polynomial graphs over GF(p^m). The synthesis and optimization is initiated from this graph based representation. The technique has been applied to minimize multipliers over all the 51 fields in GF(2^k), $k = 2 \dots 8$ in 0.18 micron CMOS technology with the help of the Synopsys® design compiler. It has also been applied to minimize combinational exponentiation circuits, and other multivariate bit- as well as word-level polynomials. The experimental results suggest that the proposed technique can reduce area, delay, and power by significant amount.

1. INTRODUCTION

Polynomials over *finite fields* and their extensions of the form GF(2^m) are crucial to certain types of crypto systems (e.g. the elliptic curve cryptography) [8], error control codes (e.g. BCH, Reed-Solomon) [26], VLSI testing [18], and DSP [6]. They also lead to designing high speed, low complexity systolic VLSI realizations [11]. In most cases the evaluation of the polynomials cannot be justified unless they are realized in high speed, low power, low complexity dedicated hardware. A polynomial over GF(2^m) constitutes addition, multiplication, and exponentiation. While addition over GF(2^m) can be realized with m 2-input EXOR gates, multiplication and exponentiation are costly in terms of area, delay, and power requirements. Most of the recent research on realizing these polynomials have focused on efficient synthesis of multipliers [2,5,17,22] and sequential [15,16] and hybrid [8] exponentiation over GF(2^m). However in general a multiple output multivariate polynomial over GF(2^m), constituting addition, multiplication, and exponentiation in different configurations, can be synthesized in hardware efficiently if appropriate factorization with simplification is first carried out. As our experimental results seem to suggest significant amount of area (up to 68 times), and well over an order of magnitude delay and power reduction can be achieved if the polynomials are synthesized and optimized as a preprocessing step before they are passed on to the industrial tools such as the Synopsys® design compiler.

[*]Partially funded by the Engineering and Physical Science Research Council (EPSRC), UK. Grant No. GR/S40855/01.

Hand synthesizing complex multiple output multivariate polynomials by factorizing, common sub-expression elimination (CSE) and algebraic simplification over GF(2^m) can be very tedious. Instead an automatic synthesis tool which can perform all of these is desirable.

Keeping this in view we propose a synthesis and optimization technique for polynomials over GF(2^m). For $m = 1$, i.e. GF(2), the proposed technique performs gate-level synthesis and optimization. For $m > 1$ it performs word-level synthesis and optimization based on the properties of GF(2^m). The technique requires that the multivariate polynomials are available in terms of their coefficients, either in a graph-based form or in terms of a network of adders or multipliers over GF(2^m), i.e. in a netlist form. If the graph-based representation is given then it performs a graph-based decomposition over GF(2^m) to obtain the netlist of adders and multipliers. However most industrial tools require AND-OR PLAs or VHDL/verilog forms as the initial specification. Therefore we also present a technique for constructing the polynomials from directed acyclic graphs (DAGs) representing the PLAs or netlists.

The underlying problem of representing polynomials over GF(p^m), p is a prime number, is the computation of the coefficients because the polynomials are determined by their coefficients. A number of techniques exist based on interpolation algorithms for small finite fields (e.g. GF(K) and $K \leq 4$), e.g. [1,14,27,28]. Although the technique of [19] seems to be applicable to large fields, it relies heavily on the size of the fields, and also for small fields a solution only exists if the interpolation points are chosen from a sufficiently large extension field [20]. Most of these techniques (e.g. [27,28]) are suitable for only the 0-polarity (i.e. only one polarity). However most practical circuits have different representations in different polarities in finite fields, e.g. certain polarities will require fewer terms or non-zero coefficients and hence fewer nodes in a graph than others for the same circuit. In this regard a technique has been presented in [24] for computing the expressions under the optimal polarity in GF(2) only. Owing to its exhaustive nature its application is limited to small circuits. Fourier like matrix based algorithms (e.g. based on coding theory, and the butterfly algorithm) exist for functions in finite fields and also generalized *Reed-Müller* forms [12]. However the size of the matrices is exponential and grows exponentially with the size of the inputs and field sizes.

Keeping these in view we present an efficient DAG based technique for computing the coefficients of polynomials over GF(p^m) in any polarity. The coefficients are stored as DAGs also as they are computed, which is the starting point of our synthesis technique.

Background. Let GF(N) denote a set of N elements, where N is a power of a prime number, with two special elements 0 and 1 representing the additive and multiplicative identities respectively, and two operators addition '$+$' and multiplication '$\cdot$'. GF(N) defines a finite field, also known as *Galois Field*, if it forms a *commutative ring* with identity over these two operators in which every element has a multiplicative inverse. Additional properties of finite fields can be found in [23, 26]. Finite fields over GF(2^m) and $m \geq 2$ can be generated with *primitive polynomials* (PP) of the form $p(x) = x^m + \sum_{i=0}^{m-1} c_i x^i$, where $c_i \in$ GF(2) [26]. For any $\alpha, \beta \in$ GF(2^m), if α and β are in their polynomial basis as $\alpha(x) = \sum_{i=0}^{m-1} \alpha_i x^i$ and $\beta(x) = \sum_{i=0}^{m-1} \beta_i x^i$, where $\alpha_i, \beta_i \in \{0,1\}$ and $0 \leq i < m$, then multiplication over GF(2^m) can be defined as $w(x) = \alpha(x) \cdot \beta(x) \mod p(x)$, where $p(x)$ represents the PP used to generate the fields [5, 23, 26]. $\alpha + \beta$, i.e. addition over GF(2^m), is the bitwise EXOR of the bit vectors corresponding to $\alpha(x)$ and $\beta(x)$.

$+$	0	1	α	β
0	0	1	α	β
1	1	0	β	α
α	α	β	0	1
β	β	α	1	0

(a) Addition

$\times$	0	1	α	β
0	0	0	0	0
1	0	1	α	β
α	0	α	β	1
β	0	β	1	α

(b) Multiplication

Figure 1: Addition and multiplication over GF(4).

Fig. 1 shows the addition and multiplication tables over GF(4). Here α and β are assumed to be elements in GF(4). For example assume that $\alpha(x) = x$, and $\beta(x) = x + 1$. $\alpha + \beta = \alpha(x) + \beta(x) = x + x + 1 = 1$. There is only one PP for generating GF(4), which is $p(x) = x^2 + x + 1$. Now $\alpha \cdot \beta = \alpha(x) \cdot \beta(x) \mod p(x) = x \cdot (x+1) \mod x^2 + x + 1 = 1$.

In this paper PPs will be considered in their decimal notation. For example the PP $p(x) = x^3 + x + 1$ in GF(8) can be represented by the bit vector $[1,0,1,1]$, which is 11 in decimal.

The following notation is used in this paper. Let $I_N = \{0, 1, \ldots, N-1\}$, and $\delta : I_N \rightarrow GF(N)$ be a one-to-one mapping with $\delta(0) = 0$, $\delta(1) = 1$ and, without loss of generality, $\delta(2) = \alpha$, $\delta(3) = \beta$, etc. Let $f(x_1, x_2, \ldots, x_k, \ldots, x_n)$ represent an n input function over GF(N). The cofactor of f w.r.t. $x_k = y$, denoted by $f|_{x_k=y}$, is $f|_{x_k=y} = f(x_1, x_1, \ldots, x_k = y, \ldots, x_n)$, i.e. $f|_{x_k=y}$ represents the fact that each occurrence of x_k in f is replaced with a y.

Given a variable x_i in GF(N) and $\pi_i \in I_N$ ($1 \leq i \leq n$, n is the number of inputs), x_i can appear in one of the N *polarities* denoted by $x_{i,\pi_i} = x_i + \delta(\pi_i)$. Certain polarities require more resources to represent a polynomial than others. As an example, let $f(x_1, x_2) = \alpha x_1 x_2$ represent a function in 0-polarity in GF(3), i.e. both x_1 and x_2 are in 0-polarity (note that $x_{i,0} = x_i$). In another polarity with x_1 in 1-polarity and x_2 in 2-polarity replacing x_1 with $x_{1,1} - 1$ and x_2 with $x_{2,2} - \delta(2) = x_{2,2} - \alpha$ and simplifying we get $f(x_1, x_2) = 1 + x_{2,2} + \alpha x_{1,1} + \alpha x_{1,1} x_{2,2}$, which contains more terms in this case. Finding an optimal polarity is a very hard problem.

Graph-Based Representation. Any function over GF(N) can be expanded in its *literal based* form as $f(x_1, \ldots, x_i, \ldots, x_n) = \sum_{e=0}^{N-1} g_e(x_i) f|_{x_i=\delta(e)}$ where $g_e(x_i) = 1 - [x_i - \delta(e)]^{N-1}$ [23]. Here $g_e(x_i)$ is called a literal over GF(N). This can be viewed as generalization of Shanon's expansion to GF(N). Such a literal based form allows Multiple-Valued Decision Diagram (MDD)-like canonic graph-based representations as shown in Fig. 2 [3, 4].

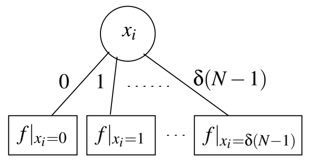

Figure 2: MODD—A literal based representation over GF(N).

Here the internal nodes represent the variables of expansion, x_i in this case, whereas the terminal nodes represent the values of the function corresponding to each value of x_i represented by the edges. However since the MDDs are defined in MIN-MAX postalgebra [13], this representation will be called *(M)ultiple (O)utput (D)ecision (D)iagram* or MODD to distinguish it from the MDDs. There are two MODD reduction rules [3]: (a) If all the N children of a node v point to the same node w, then delete v and connect the incoming edge of v to w. (b) Share equivalent subgraphs. A reduced MODD can be further optimized and normalized based on additional two rules mentioned in [4]. An MODD which is reduced by all 4 rules will be called a *(Z)ero suppressed and (N)ormalized MODD* or ZNMODD [4].

This paper is organized as follows. Section 2 presents a technique for computing and storing coefficients over GF(p^m). Section 3 presents a synthesis and optimization technique based on the DAG based representation of Section 2. Finally, Section 4 presents experimental results.

2. COMPUTATION OF COEFFICIENTS

Given a function $f(x_n, x_{n-1}, \ldots, x_1)$ in GF(N) (N is a power of a prime number) and a polarity number t, f can be represented in the following canonical multivariate polynomial form [23].

$$f(x_n, x_{n-1}, \ldots, x_1) = \kappa_{0,t} + \kappa_{1,t}\tilde{x}_1 + \kappa_{N,t}\tilde{x}_2 + \cdots + \kappa_{i,t}\tilde{x}_{j_k}^{i_k}\tilde{x}_{j_{k-1}}^{i_{k-1}} \cdots \tilde{x}_{j_1}^{i_1} + \cdots + \kappa_{N^n-1,t}\tilde{x}_n^{N-1}\tilde{x}_{n-1}^{N-1} \cdots \tilde{x}_1^{N-1}.$$

Here $\kappa_{i,t}$ represents the coefficient of the ith term containing k number of variables in polarity number t, where both i and t are defined in the radix-N number system as $i = i_k N^{j_k} + i_{k-1} N^{j_{k-1}} + \cdots + i_1 N^{j_1}$ and $t = \pi_n N^{n-1} + \pi_{n-1} N^{n-2} + \cdots + \pi_1$.

Further, each π_q represents the polarity of the variable x_q and $1 \leq q \leq n$, i.e. $x_{q,\pi_q} = x_q + \delta(\pi_q)$. Here $\tilde{x}_l$ represents the fact that x_l is in any one of the polarities $\{0, 1, \ldots, N-1\}$, as determined by t. For example, the term associated with the coefficient $\kappa_{i,t}$ is $\tilde{x}_{j_k}^{i_k}\tilde{x}_{j_{k-1}}^{i_{k-1}} \cdots \tilde{x}_{j_1}^{i_1}$, where the polarity of each variable is determined by t. Each coefficient can be determined by the following.

Theorem 1. *[4] Let $f(x_n, x_{n-1}, \ldots, x_1)$ represent an n variable function in GF(N). Its ith coefficient corresponding to a term containing k number of variables in the polarity number t is*

$$\kappa_{i,t} = (-1)^k \sum_{x_{j_k}=\delta(0)}^{\delta(N-1)} \sum_{x_{j_{k-1}}=\delta(0)}^{\delta(N-1)} \cdots \sum_{x_{j_1}=\delta(0)}^{\delta(N-1)} \theta_{i,t}(\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_n)$$
$$f(x_{j_k}, x_{j_{k-1}}, \ldots, x_{j_1}, -\delta(\pi_l), -\delta(\pi_{l-1}), \ldots, -\delta(\pi_1)), \quad (1)$$

where $\delta(\pi_l), \delta(\pi_{l-1}), \ldots, \delta(\pi_1)$ *are the polarities of the variables* $x_l, x_{l-1}, \ldots, x_1$ *as determined by* t.

Here $\theta_{i,t}$ is defined in [4, 23]. In Eq. (1) those variables which are (not) assigned any value will be called assigned (unassigned) variables.

In many cases the MODD (or MDD) may be available as a part of existing resources during a synthesis, verification or simulation process, in which case the coefficients can be derived from the graphs by means of an efficient path oriented technique given in [4] based on Theorem 1. We call this technique 'CompByPath'. For many benchmarks it executed quickly. However applying this algorithm for determining *all* the coefficients in large fields can render the process slow since it could involve repeated search through the MODDs. In this paper we further improve this technique for significant speedup in large fields as illustrated in the following example.

Let $f(x_1,x_2,x_3)$ be a function over GF(4) with x_1, x_2, and x_3 in polarities π_1, π_2, and π_3 respectively. The coefficient $\kappa_{2,t}$ of the term x^2_{3,π_3} can be computed as $\kappa_{2,t} = \sum_{x_3=0}^{\beta}(x_3 + \delta(\pi_3))f(\delta(\pi_1), \delta(\pi_2), x_3)$ from Theorem 1 using Algorithm CompByPath [4]. Let us denote $\mathcal{PC} = (x_3 + \delta(\pi_3))f(\delta(\pi_1), \delta(\pi_2), x_3)$, which we have termed as the *partial coefficient*. If during this computation $\mathcal{PC}$ is stored (e.g. in a hash table) for the relevant values of x_3 from the MODD, then the coefficient of x_{3,π_3} can be computed as $\kappa_{1,t} = \sum_{x_3=0}^{\beta}(x_3 + \delta(\pi_3))^2 f(\delta(\pi_1), \delta(\pi_2), x_3) = \sum_{x_3=0}^{\beta} \mathcal{PC} \cdot (x_3 + \delta(\pi_3))$, i.e. by multiplying the values of $x_3 + \delta(\pi_3)$ by the stored values of $\mathcal{PC}$ and then adding them together without having to revisit the MODD. In this way any coefficient over GF(N) with smaller exponents can be determined from those with larger ones without revisiting the MODDs as explained below.

Let us represent each unassigned variable with 1 and each assigned variable with 0. This results in a set of binary vectors of dimension n. The set of all such binary vectors defines a lattice over Boolean algebra. The Greatest Lower Bound (GLB) of this lattice is $[00\cdots0]$ (i.e. all variables assigned), while the Least Upper Bound (LUB) is $[11\cdots1]$ (i.e. all variables unassigned). Clearly an n-variable function over GF(N) will have 2^n points in the lattice. For each 1 in the vector corresponding to a point in the lattice, apart from the GLB, the corresponding variable will have N different exponents in the term ranging from 1 to $N-1$ (each point will correspond to a single coefficient over GF(2)). The algorithm proceeds as follows. Each point pt in the lattice is considered once from $[00\cdots0]$ to $[11\cdots1]$. The point $[00\cdots0]$ is determined by Theorem 1 directly. For example for a 4-input function over GF(N) assume $pt = [0101]$. This corresponds to the exponents $\{(0,N-1,0,N-1), (0,N-1,0,N-2), \ldots, (0,0,0,0)\}$. The exponents are ordered so that they differ in one place and by exactly one and processed from the highest to the lowest exponent. The coefficient corresponding to the highest exponent, i.e. $(0,N-1,0,N-1)$, is determined by Algorithm CompByPath. The rest of the coefficients are determined incrementally as outlined previously without revisiting the MODD, e.g. the coefficient of $(0,N-1,0,N-2)$ from $(0,N-1,0,N-1)$, etc.

Storing the Coefficients. The coefficients are stored as they are computed in a reduced and shared DAG called the (S)hared (G)alois (P)olynomial (D)ecision (D)iagram or SGPDD. The basic idea follows. Any function in GF(N) can be represented (expanded) based on the following polynomial form: $f(x_1, \ldots, x_i, \ldots, x_n) = h_{\pi_i,0} + \sum_{r=1}^{N-1} x^r_{i,\pi_i} h_{\pi_i,r}$. Here $h_{\pi_i,j}$ is the coefficient corresponding to the jth term in π_i polarity. This can be represented by means of a graph as shown in Fig. 3. Here the terminal nodes represent the coefficients over GF(N) while the edges represent the exponents of

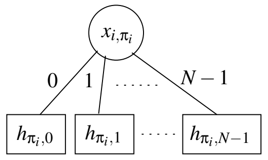

Figure 3: Representing and storing coefficients.

x_i. The GPDD can be reduced as follows: Given two nodes v and w such that $child_i(v) = w$ and $0 \le i \le N-1$, (a) if all the nonzero edges of w point to the zero terminal node, then reconnect $child_i(v)$ to $child_0(w)$ and delete w, (b) Share equivalent subgraphs. A GPDD reduced based on these two rules can be shown to be canonic and minimal under a fixed variable ordering. Note that the MODDs represent functions over GF(N) in their literal based forms whereas the GPDDs represent them in their polynomial forms based on their coefficients. An efficient GPDD reduction algorithm having linear complexity in the number of nodes has been developed for arbitrary GF(N) and any polarity. This algorithm is analogous to the BDD reduction algorithm [7] with the exception that the reduction rule has been replaced with rule-(a) above while the sharing rule remains the same and, obviously, the number of children has been extended to N. Note that the GPDD reduces to the functional decision diagram (FDD) over GF(2) [25]. It also has integer interpretation as TED [21].

Fig. 4(a) shows a reduced and shared GPDD for the polynomial $f(a,b,c) = ac + b^2c$ in GF(4) in 0-polarity. Each internal node has 4 children. The edges are labeled 0 to 3. However edges terminating at the 0 terminal node are not labeled for brevity.

3. SYNTHESIS AND OPTIMIZATION

Once the SGPDD is obtained circuits are synthesized by decomposing and factoring the SGPDD based on finding *cuts* within the GPDD. A cut is a partitioning of the nodes in the SGPDD into two sets T and B, where T contains internal nodes and the root and B contains external, internal, and the last nodes, i.e. internal nodes which have the external nodes as their children. A cut passes through an SGPDD horizontally and does not cross an edge more than once. Effectively a cut can factorize an SGPDD realizing a function f in GF(2^m) as $f = D \cdot Q + R$. Cut based algorithms have been used for synthesis in the Boolean domain, e.g. [9]. In this paper we quickly factorize a polynomial over GF(2^m) based on cuts on their GPDDs to construct an expression DAG based multiple output shared netlist. The netlist constitutes two types of nodes: internal nodes which can either be GF(2^m) adders or multipliers, or external nodes which can only be constants and variables in GF(2^m). The internal nodes can have two children. Fig. 5(d) shows the netlist for the expression $ab^2 + bc$. The netlists are further synthesized based on additional factorization and optimization.

3.1 Decomposing from GPDD

Given a variable x and a GPDD for a function f, there are two types of decomposition possible: (i) multiplicative, which represents f as $f = X \times Y$, and (ii) additive, which represents f as $f = W + Z$. Here X, Y, W, and Z are GPDDs. To perform a decomposition a cut is performed above the nodes representing x. Let v_x be a node representing x. To obtain the multiplicative decomposition all the paths in the original GPDD from nodes above the cut leading to v_x are reconnected to the terminal node 1 and the result is reduced. This gives X. Y is simply the GPDD rooted at v_x. To obtain the additive decomposition all the paths in the original GPDD from nodes above

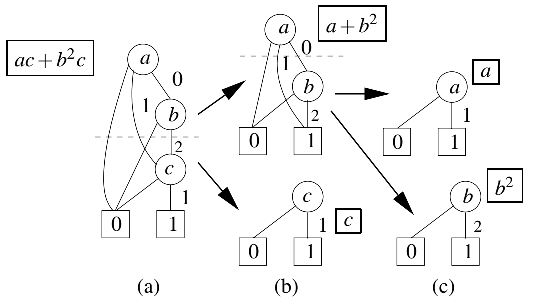

Figure 4: Factorizing from GPDD.

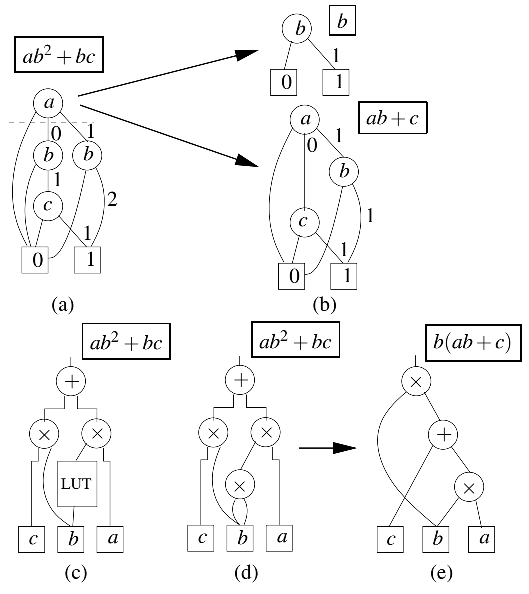

Figure 5: Constructing the expression tree.

the cut leading to v_x are reconnected to the terminal node 0 and the result is reduced. This gives W. Z is obtained by reconnecting all the paths from nodes above the cut that do not pass through v_x to the terminal node 0 and reducing the result. The proof for this reasoning is straight forward and has been left out for brevity. The following example demonstrates the basic idea.

In the GPDD of Fig. 4(a) the cut is shown with the horizontal broken line above the node c. Note that cuts can also be made above node b. However the cut above c has been made because it is a *dominator* node [9], i.e. all paths ending in the terminal x node (in this case $x = 1$) passes through node c. Given a dominator node v, v can be trivially factored out of a GPDD by reconnecting all the paths leading to v to the terminal node 1 and detaching the subgraph rooted at v as another GPDD. This results in a multiplicative decomposition over GF(2^m). After cutting the GPDD each of the component is reduced. While the bottom GPDD in Fig. 4 cannot be decomposed further, the top one can be decomposed as two GPDDs: one rooted at a, and the other rooted at b (Fig. 4(c)). Here the cut is performed above node b, which results in an additive decomposition over GF(2^m). These nodes are not further decomposed but instead added to the netlist as $c \times (a+b^2)$, which requires one multiplier and one adder over GF(2^m). The exponent b^2 can be implemented in two ways: either using an additional multiplier for the exponentiation, or using an m-input m-output look-up table (LUT) since we are dealing with GF(2^m). Our tool can do either depending on which option is given.

A more general case appears in Fig. 5. Here the function considered is ab^2+bc. Decomposition of expressions like this cannot be done based on the approach of [9] owing to the presence of the exponent. Fig. 5(b) shows how a multiplicative decomposition is carried out if the cut is performed above the nodes b.

In this paper we perform factorization of exponents on the netlists, which we obtain by using a fast greedy heuristic algorithm as outlined in Fig. 6.

Much of the algorithm is self explanatory. The algorithm decomposes a function f as $f = D \cdot Q + R$. In Line 8 the list of parents is computed for the GPDD rooted at *root*. Lines 10–17 are executed for nodes with at least one parent.

The cut is performed above the node Q. Lines 11 and 13 perform additive decomposition by reconnecting all paths coming from nodes above the cut not leading to (Line 11) and leading to (Line 13) Q to the zero terminal node. Line 12 performs multiplicative decomposition. These steps are repeated for each of the components Q, D, and R. Line 20 performs netlist optimization as outlined in Section 3.3. Note that no assumption is made regarding the word-size or the polarity.

Fig. 4 shows the decomposition obtained from the GPDD of Fig. 4(a) using this algorithm. Fig. 5(c) (exponent realized with LUT) and (d) (exponent realized with repeated multiplication) shows the resulting netlist after decomposing the GPDD of Fig. 5(a).

3.2 Factorizing Netlists

Common factors are determined by walking through chains of multipliers following chains of adders. A chain of adders is a non-empty ordered list of adders $\langle A_i, A_{i+1}, \ldots, A_{i+r}\rangle$ such that A_i forms a chain if $r = 0$; otherwise A_{j+1} appears as one of the inputs to A_j for $i \le j < i+r$. A chain of multipliers is an ordered list of multipliers $\langle M_k, M_{k+1}, \ldots, M_{k+s}\rangle$ such that M_{j+1} appears as one of the inputs to M_j for $k \le j < k+s$. A chain of multipliers can be empty, in which case a term (i.e. a node u or a subgraph rooted at u) is considered as a chain. Let $C_{A_1} = \langle A_i, A_{i+1}, \ldots, A_{i+r}\rangle$ and $C_{A_2} = \langle A_j, A_{j+1}, \ldots, A_{j+s}\rangle$ be two chains of adders. Let $C_{M_1} = \langle M_k, M_{k+1}, \ldots, M_{k+t}\rangle$ and $C_{M_2} = \langle M_l, M_{l+1}, \ldots, M_{l+w}\rangle$ be two chains of multipliers. Let M_k and M_l be one of the inputs to A_{i+r} and A_{j+s} respectively. Let X be a subgraph rooted at X such that X appears as one of the inputs to M_{k+t} and M_{l+w}. If $A_i = A_j$, then X is factorizable. The proof is straight forward and has been left out for brevity. The factorization is carried out as illustrated in Fig. 7. Fig. 7(a) shows the structure $(AX+BX)$ which can be factored as $X(A+B)$ as shown in Fig. 7(b) and (c). Fig. 7(d) shows a more general structure of the form $Z = ((AX+Y)+BX)$. Clearly X is factorizable. To factorize X Z is restructured as $Z = ((AX+BX)+Y)$ (Fig. 7(e)) and then the factorization is carried out as $Z = ((X(A+B))+Y)$ (Fig. 7(f)). The structure within the circle in Fig. 7(e) is the network of Fig. 7(d) after its pointers have been readjusted. After restructuring the circuits the netlist optimization technique of Section 3.3 is applied to obtain the final circuits. For example factorizing the netlist of Fig. 5(d) yields the network of Fig. 5(e). This result is the same as decomposing the GPDD shown in Fig. 5(a) and (b).

The algorithm for factorization proceeds by trying out all possible factorizations, and only stops when there are no more terms which can be factored out. Note that no assumption regarding the word-size or the polarity has been made, i.e. the same approach works for bit-level as well as word-level circuits in any polarity.

3.3 Optimizing Netlists

Netlists are optimized by processing them recursively from the external (i.e. variable or constant) nodes towards the root node. Each

```
Algorithm DecompGpdd{
    Let NodeQ be a queue initially empty;
    Let root be the root of the GPDD;
    Let EtRoot be the root of the netlist;
    Add root to the netlist and push root on to NodeQ;
    do{
        Remove head of NodeQ and assign to root;
        Construct parent list of root;
        if(ParentList > 0){
            Q = node with the least number of parents;
            D = AssignPassAroundZ(Q);
            Reconnect all edges in D leading to Q to 1;
            R = AssignPassThroughZ(Q);
            D = reduce(D);
            R = reduce(R);
            Add Q, D, and R to the netlist;
            Push Q, D, and R on to NodeQ;
        }
    }while(NodeQ is not empty);
    EtRoot = optimize(EtRoot);
}
```

Figure 6: A GPDD decomposition algorithm.

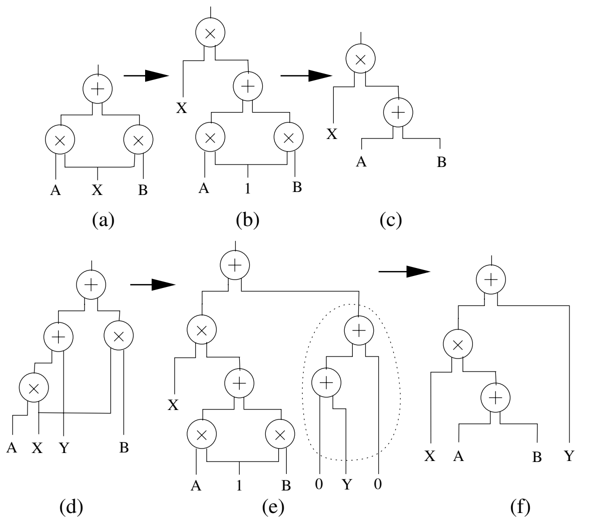

Figure 7: Factorizing from netlist.

node is visited exactly once. For each node u the following is done. If u is already processed then its reference is returned so that it can be shared; otherwise its information is stored in a hash table for sharing. If u is an internal (i.e. GF(2^m) multiplier/adder) node, then let v_0 and v_1 be its two children. If both v_0 and v_1 are constants, then replace u with v_0 op v_1 where op is either addition or multiplication over GF(2^m), i.e. perform *constant propagation*. Replace u with v_i ($i \in \{0,1\}$), if $u = v_i \times 1$ or $u = v_i + 0$. Replace u with 0, if $u = v_i \times 0$ or $u = v_i + v_i$. Note that this algorithm is analogous to the BDD reduction algorithm of [7], which can be argued to be optimal under a fixed variable ordering. Analogously this algorithm can be shown to be optimal w.r.t. two input addition and multiplication over GF(2^m) under a fixed netlist transformation. Also note that it can be applied to bit-level as well as word-level correctly with either repeated multiplication or LUT based exponentiation and in any polarity. This algorithm can be applied for single as well as multiple output functions.

For example given the netlists of Fig. 7(b) and (e) this algorithm will yield the netlists of Fig. 7(c) and (f) respectively.

3.4 Exponentiation

Exponentiation is carried out in two ways: (i) using the repeated multiplication rule, (ii) using LUTs, depending on which option is given. If rule (i) is specified, then exponents are generated as follows: square and multiply incrementally, e.g. given x^4, x^5 and x^{11}, generate x^4 using square and multiply, $x^5 = x^4 \times x$ and $x^{11} = x^5 \times x^6$, where $x^6 = x^5 \times x$. This algorithm is called 'Exp'.

If rule (ii), i.e. the LUT option is given, then polynomials of the form $p(x) = \sum_{i=0}^{2^m-1} c_i x^i$, where $c_i \in$ GF(2^m), are generated as a single m-input m-output LUT. For example given an expression $a^3(\beta b^9 + b) + c(\beta b^9 + b)$ in GF(16), it is first factored as $(\beta b^9 + b)(a^3 + c)$. Then the result is implemented using two 4-input 4-output LUTs, one for each $\beta b^9 + b$ and a^3, one multiplier and one adder over GF(16).

3.5 Overall Algorithm

The algorithm proceeds as follows: (i) Store initial specification as reduced and shared MODD, (ii) apply the technique of Section 2 to compute and store the coefficients as reduced and shared GPDDs, (iii) apply the techniques of Section 3.1 to decompose the GPDDs into the netlist, (iv) apply the technique of Section 3.2 to further decompose and optimize the netlist. During this process the technique of Section 3.3 is applied between Steps (iii) and (iv) to obtain an initial, and also after Step (iv) to obtain the final result. The best result is retained. The algorithm can be initiated at any point, e.g. if the GPDD is available then we can start from Step (iii).

4. EXPERIMENTAL RESULTS

The techniques presented in this paper have been implemented in Gnu C++ 3.2.2-5 on a computer with 640MB RAM and a 600MHz Athlon processor running RedHat Linux with kernel-2.4.20-43.9. The Synopsys® design compiler was run on a dual processor Pentium 4 Linux machine with 2GB RAM and kernel-2.4.21-20.EL. The benchmarks were stored as two-level AND-OR PLAs to enable us to determine how effective the proposed technique is in optimizing area, power, and delay. Also the Synopsys® design compiler understands this format. The PLAs were read into MODDs, and the algorithm of Section 3.5 was applied from Step (i). After optimization the results were saved in the VHDL format, which were passed to the Synopsys® design compiler (power was estimated with the Synopsys® power compiler). The PLAs were also passed directly to the Synopsys® compiler for optimizing without the aid of the proposed technique.

Unless otherwise specified the tool assumes the following primitive polynomials by default for generating the fields: $(2,7),(3,11),(4,19),(5,37),(6,67),(7,137),(8,285)$. Here the first number m in the ordered pair (m,d) is the bit width and determines the field size, i.e. GF(2^m), and the second number d is the decimal representation of the primitive polynomial.

We have minimized multipliers over GF(2^m) ($2 \le m \le 8$) for all the 51 primitive polynomials in 0.18 micron CMOS technology. Table 1 shows results for the 8-bit multipliers. Column 1 represents the primitive polynomials, while Column 2 represents the area, delay, and power reported by the Synopsys® design compiler without the aid of the proposed technique. The column with the heading "Proposed Technique" shows the result of applying the proposed technique first, and then applying the Synopsys® compiler on the resulting VHDL files. The letters 'a' and 'm' represent 2-input EXOR and AND gates respectively. Here area, delay, and power are in 10^{-6} mm^2, nano seconds, and mW respectively. Power was estimated at 1.8V. Significant area, delay, and power reduction is observable (as much as 57 times for polynomial 487). Clearly the number of AND gates is m^2 for all the cases. The number of EXOR gates

Table 1: Multipliers over GF(2^8) for all the 16 PPs.

Prim	Synopsys® only	Proposed Technique	
Poly	(area,delay,power)	(a,m)	(area,delay,power)
285	(141689.3,18.8,130.9)	(87,64)	(2628.9,1.6,4.2)
299	(152603.3,14.6,125.7)	(85,64)	(2609.5,1.7,4.1)
301	(132595.5,15.0,125.4)	(84,64)	(2580.5,1.5,4.0)
333	(140896.3,15.2,129.3)	(84,64)	(2574.0,2.0,4.1)
351	(135107.6,17.9,121.5)	(100,64)	(2757.9,1.5,4.3)
355	(143195.4,17.4,126.0)	(88,64)	(2612.7,1.8,4.2)
357	(144472.7,14.8,133.5)	(84,64)	(2548.2,1.5,4.0)
361	(135227.9,16.3,125.6)	(88,64)	(2628.9,1.7,4.2)
369	(132405.8,17.6,130.2)	(89,64)	(2722.4,1.9,4.3)
391	(130577.4,15.9,126.5)	(79,64)	(2441.8,2.2,3.8)
397	(139992.8,14.4,122.3)	(87,64)	(2532.1,2.0,4.0)
425	(141879.6,15.6,118.9)	(94,64)	(2690.2,2.0,4.3)
451	(139612.7,17.6,123.9)	(79,64)	(2493.4,1.8,3.9)
463	(141228.2,17.8,122.9)	(90,64)	(2661.1,1.9,4.1)
487	(150703.6,16.7,138.8)	(92,64)	(2632.1,1.9,4.1)
501	(129935.6,16.0,119.2)	(104,64)	(2906.3,2.3,4.7)

varied from approximately m^2-1 for trinomials, e.g. 19 ($m=4$, 15 EXORs), 25 ($m=4$, 15 EXORs), 37 ($m=5$, 25 EXORs), 67 ($m=6$, 35 EXORs), 131 ($m=7$, 48 EXORs), etc., to m^2+k, where k is a constant, for pentanomials and polynomials with more than 5 terms. As compared with [22], which reports $2m^2-1$ 2-input AND gates and $2m^2-3m+1$ EXOR gates, the proposed technique produced better results. For the 8-bit multiplier this technique reports maximum area of 0.002906 mm^2, whereas [22] reported 0.0128 mm^2, i.e. about 4.4 times better (5.2 times better for polynomial 391). Also, it reports about 2 times reduced delay. For the 4-bit case the proposed technique reports about 0.000522 mm^2 area with the PP 25, i.e. over 5 times better. It is not possible to directly compare this technique with [2,5,17] because these techniques have reported only theoretical results, with m^2 2-input AND gates and m^2-1 (for trinomials) to m^2+k EXOR gates depending on the number of terms in the polynomials and their positions without any implementation. However we were able to compare the examples given in [2,5]. For examples [5] reported 18 EXOR and 16 AND gates for the GF(2^4) multiplier in the primitive polynomial 25, whereas this technique reports 15 EXOR and 16 AND gates. As compared with [2] the proposed technique reported similar result for the polynomial 171 over GF(2^7), i.e. 64 EXOR and 49 AND gates. Note that these results closely approximate the theoretical results reported by these techniques despite the fact that the proposed technique is a heuristic synthesis algorithm for polynomials, whereas these techniques are designed for hand synthesizing the multipliers over GF(2^m).

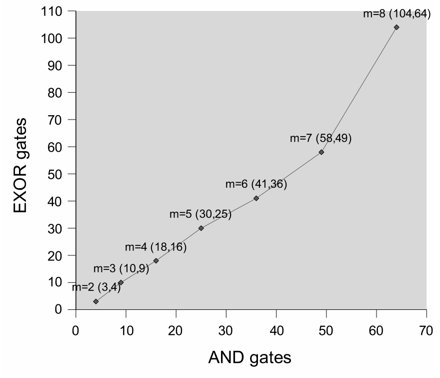

Figure 8: Gate count for ab^2.

Fig. 8 shows the result of applying the technique for optimizing the term ab^2 in GF(2^m), where $2 \leq m \leq 8$. The total number of AND gates is exactly m^2. The total EXOR gate count is approximately m^2+m up to $m=7$. For $m=8$ the EXOR gate count worsens. Note that techniques such as [10], which are tailored for hand synthesizing ab^2, have reported total EXOR and AND gate counts of $(m+1)^2$ each.

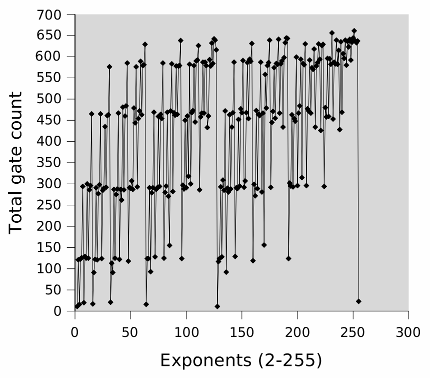

Figure 9: Variation of 2-input gates with exponent over GF(2^8).

Fig. 9 shows how the total 2-input gate count varies with the exponent x^y, where $2 \leq y \leq 255$, over GF(2^8). There are distinct regions noticeable which seems to be repeating with the exponents. For exponents 2, 4, 8, 16, 32, and 128 only 2-input EXOR gates were required with 11 being the minimum (for exponents 2 and 128) and 21 being the maximum (for exponent 32). The hardest exponent was 251, which required 410 and 251 2-input EXOR and AND gates respectively, i.e. 661 gates in total. Exponent 255 requires fourteen 2-input EXOR gates, and nine 2-input AND gates, which can be realized with a single 8-input OR gate.

Table 2: Word-level synthesis and optimization.

Field (k,pp)	Polynomial	Exp (a,m)	LUT (a,m,LUT)	Gate Level (a,m)
(6,67)	$a^{13}+b^{11}$	(1,10) [(356,360)]	(1,0,2) [(134,83)]	(136,83)
(6,67)	$a^6b^7+a^7b^{10}$	(1,11) [(391,396)]	(1,2,4) [(289,207)]	(1295,631)
(6,67)	$a^{59}b^{35}+a^{29}b^{63}$	(1,27) [(951,972)]	(1,2,4) [(320,247)]	(1300,745)
(5,37)	$a^{30}b^{29}c+b^{29}c^{17}$	(1,22) [(555,550)]	(1,2,3) [(189,154)]	(1238,748)

Table 2 shows the results for some sample multivariate polynomials. The polynomials have been synthesized at the word as well as at the gate-level. At the word-level Exp and the LUT based techniques have been applied for the generation of exponents. The symbols 'k', 'pp', 'a' and 'm' represent the word size, primitive polynomial, total number of 2-input adders (EXOR gates in GF(2)) and multipliers (AND gates in GF(2)) respectively. The figures within '[]' represent the maximum number of 2-input EXOR and AND gates required to implement the adders, multipliers, and LUTs. This figure is obtained by synthesizing the LUTs at the gate level with the proposed technique, and then adding the total number of 2-input EXOR and AND gates required for the LUTs, adders, and multipliers (Table 1). Clearly the LUT based algorithm is winning out, while Exp is in the second place. However LUTs suffer from exploding in size quite rapidly, unless they are saved in a DAG based form, e.g. the BDD, as they are created.

We have also applied the proposed technique on ab^y for $2 \leq y \leq 255$ over GF(2^8) at the gate-level. The minimum area is 148.7×10^{-6} mm^2 for ab^{255}, and the maximum area is 60902×10^{-6} mm^2 for ab^{251}. We could not compare all the cases produced by the Synopsys® compilers alone since for most of the cases it was taking far too long without the aid of the proposed technique, e.g. for ab^{79} we had to terminate the compiler after 51 hours. We could apply the Synopsys® compilers for certain cases, e.g. for ab^{68} it reported 790785×10^{-6} mm^2 area, and 24.1 ns delay without the proposed technique, whereas with the proposed technique it reported 11531×10^{-6} mm^2 area and 3.58 ns delay, i.e. about 68 times area and nearly an order of magnitude delay improvement respectively. If the proposed technique is applied at the word level (8-bit word size), then for ab^{251} with b^{251} implemented as LUT the gate count reduces from 2280 EXOR, and 1275 AND gates down to a maximum of 497 EXOR and 315 AND gates. Here the multiplier and the LUT are minimized at the gate-level.

Note that in all the cases many 2-input EXOR gates were of the form $1+x$, which is a single NOT gate but still counted as an EXOR gate.

5. CONCLUSIONS

In this paper we presented a heuristic synthesis and optimization technique for polynomials in GF(2^m) for gate- as well as word-level. The experimental results suggest that this technique can significantly reduce area, delay, and power. Also this technique can closely match techniques tailored for hand synthesizing circuits in GF(2^m). Therefore we can conclude that if this technique is used in conjunction with those techniques, then near optimal circuits can be designed for polynomials over GF(2^m).

6. REFERENCES

[1] A. Dur and J. Grabmeier. Applying Coding Theory to Sparse Interpolation. *SIAM J. Computing*, 22(4):695–703, Aug. 1993.

[2] A. Halbutogullari and Çetin K. Koç. Mastrovito Multiplier for General Irreducible Polynomials. *IEEE Trans. Comput.*, 49(5):503–518, May 2000.

[3] A. Jabir and D. Pradhan. MODD: A New Decision Diagram and Representation for Multiple Output Binary Functions. In *Des. Automat. Test. in Europe (DATE'04)*, pages 1388–1389, Paris, France, Feb. 2004.

[4] A. Jabir and D. Pradhan. An Efficient Graph Based Representation of Circuits and Calculation of Their Coefficients in Finite Field. In *Proc. Int. Workshop on Logic and Synth. (IWLS'05)*, pages 218–225, California, USA, June 2005.

[5] A. Reyhani-Masoleh and M.A. Hasan. Low Complexity Bit Parallel Architectures for Polynomial Basis Multiplication over GF(2^m). *IEEE Trans. Comp.*, 53(8):945–959, Aug. 2004.

[6] R. Blahut. *Fast Algorithms for Digital Signal Processing*. Addison-Wesley, Reading, Mass., 1984.

[7] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, C–35(8):677–691, Aug. 1986.

[8] C. Paar, P. Fleischmann, and P. Soria-Rodriquez. Fast Arithmetic for Public-key Algorithms in Galois Fields with Composite Exponents. *IEEE Trans. Comp.*, 48(10):1025–1034, Oct. 1999.

[9] C. Wang, V. Singal, and M. Ciesielski. BDD Decomposition for Efficient Logic Synthesis. In *Int. Conf. Comput. Aided Design (ICCAD)*, pages 626–631, 1999.

[10] C.H. Liu, N.F. Huang, and C.Y. Lee. Computation of ab^2 Multiplier in GF(2^m) Using an Efficient Low Complexity Cellular Architecture. *IEICE Trans. Fund. Electron. Comm. Comp. Sci.*, E83-A(12):2657–2663, 2000.

[11] C.H. Wu, C.M. Wu, M.D. Sheih, and Y.T. Hwang. High-Speed, Low-Complexity Systolic Design of Novel Iterative Division Algorithm in GF(2^m). *IEEE Trans. Comput.*, 53:375–380, Mar. 2004.

[12] D.K. Pradhan and A.M. Patel. Reed-Müller Like Canonic Forms for Multivalued Functions. *IEEE Trans. Comp.*, C-24(2):206–210, Feb. 1975.

[13] D.M. Miller and R. Drechsler. On the Construction of Multiple-Valued Decision Diagrams. In *Proc. 32nd ISMVL*, pages 245–253, Boston, USA, 2002.

[14] D.Y. Grigoriev and M. Karpinski. Fast Parallel Algorithms for Sparse Multivariate Polynomials Over Finite Fields. *SIAM J. Computing*, 19(6):1059–1063, Dec. 1990.

[15] H. Wu and M.A. Hasan. Efficient Exponentiation of a Primitive Root in GF(2^m). *IEEE Trans. Comp.*, 46(2):162–172, Feb. 1997.

[16] J.C. Jeon and K.Y. Yoo. Low Power Exponent Architecture in Finite Field. *IEE Proc. Part-E*, 152(6):573–578, Dec. 2005.

[17] J.L. Imaña, J.M. Sánches, and F. Tirado. Bit Parallel Finite Field Multipliers for Irreducible Trinomials. *IEEE Trans. Comp.*, 55(5):520–533, May 2006.

[18] M. Abramovici, M. Breuer, and A. Friedman. *Digital Systems Testing and Testable Design*. IEEE Publications, 1994.

[19] M. Ben-Or and P. Tiwari. A Deterministic Algorithm for Sparse Multivariate Polynomial Interpolation. In *Proc. 20th Symp. Theory of Computing*, pages 301–309, Apr. 1988.

[20] M. Clausen, A. Dress, J. Grebmeier, and M. Karpinski. On Zero-Testing and Interpolation of k-Sparse Polynomials over Finite Fields. *Theoretical Comp. Sci.*, 84(2):151–164, Jan. 1991.

[21] M.J. Ciesielski, P. Kalla, Z. Zeng, and B. Rouzeyere. Taylor Expansion Diagrams: A Compact, Canonical Representation with Applications to Symbolic Verification. In *Design Automation and Test in Europe*, Mar. 2002.

[22] N. Iliev, J.E. Stine, and N. Jachimiec. Parallel Programmable Finite Field GF(2^m) Multipliers. In *Proc. IEEE Comp. Soc. Annual Symp. VLSI Emerging Trends (ISVLSI'04)*, pages 299–302, Feb. 2004.

[23] D. Pradhan. A Theory of Galois Switching Functions. *IEEE Trans. Comp.*, C–27(3):239–249, Mar. 1978.

[24] T. Sasao and F. Izuhara. Exact Minimization of FPRMs Using Multi-Terminal EXOR TDDs. In T. Sasao and M. Fujita, editors, *Representations of Discrete Functions*, pages 191–210. Kluwer Academic Publishers, 1996.

[25] U. Kebschull and W. Rosenstiel. Efficient graph-based computation and manipulation of functional decision diagrams. In *Proc. European Design Automation Conf.*, pages 278–283, Feb. 1993.

[26] S. Wicker. *Error Control Systems for Digital Communication and Storage*. Prentice Hall, Englwood Cliffs, N.J., 1995.

[27] Z. Zilic and Z. Vranesic. A Multiple-Valued Reed-Müller Transform for Incompletely Specified Functions. *IEEE Trans. Comp.*, 44(8):1012–1020, Aug. 1994.

[28] Z. Zilic and Z.G. Vranesic. A Deterministic Multivariate Interpolation Algorithm for Small Finite Fields. *IEEE Trans. Comput.*, 51(9):1100–1105, Sept. 2002.