Optimization of Polynomial Datapaths Using Finite Ring Algebra

SIVARAM GOPALAKRISHNAN and PRIYANK KALLA
University of Utah

This article presents an approach to area optimization of arithmetic datapaths at register-transfer level (RTL). The focus is on those designs that perform polynomial computations (ADD, MULT) over finite word-length operands (bit-vectors). We model such polynomial computations over m-bit vectors as algebra over finite integer rings of residue classes Z_{2^m}. Subsequently, we use the number-theoretic and algebraic properties of such rings to transform a given datapath computation into another, bit-true equivalent computation. We also derive a cost model to estimate, at RTL, the area cost of the computation. Using the transformation procedure along with the cost model, we devise algorithmic procedures to search for a lower-cost implementation. We show how these theoretical concepts can be applied to RTL optimization of arithmetic datapaths within practical CAD settings. Experiments conducted over a variety of benchmarks demonstrate substantial optimizations using our approach.

Categories and Subject Descriptors: B.5.1 [**Register-Transfer-Level Implementation**]: Design—*Datapath design*

General Terms: Algorithms

Additional Key Words and Phrases: High-level synthesis, polynomial datapaths, arithmetic datapaths, modulo arithmetic, finite ring algebra

ACM Reference Format:

Gopalakrishnan, S. and Kalla, P. 2007. Optimization of polynomial datapaths using finite ring algebra. ACM Trans. Des. Automat. Electron. Syst. 12, 4, Article 49 (September 2007), 30 pages. DOI = 10.1145/1278349.1278362 http://doi.acm.org/10.1145/1278349.1278362

1. INTRODUCTION

RTL descriptions of integer datapaths that implement polynomial arithmetic are found in many practical applications, such as digital signal processing (DSP) for audio, video, and multimedia applications [Mathews and Sicuranza 2000;

This work has been supported in part by a grant from the US National Science Foundation Faculty Early Career (CAREER) Development Award, CCF-546859.

Authors' address: S. Gopalakrishnan and P. Kalla, Department of Electrical and Computer Engineering, University of Utah, 50 S. Central Campus Dr., Rm. 3280 MEB, Salt Lake City, UT 84112; email: {sgopalak, kalla}@ece.utah.edu.

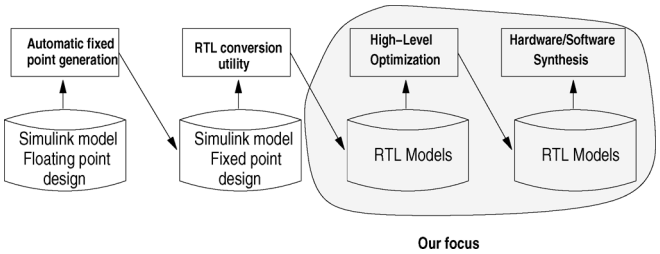

Fig. 1. Typical design flow for arithmetic datapath-intensive applications

Peymandoust and DeMicheli 2003]. The growing market for such applications requires sophisticated CAD support for design, optimization, and synthesis.

Arithmetic datapath, intensive designs implement a sequence of ADD, MULT type of algebraic computations over bit-vectors; hence they are generally modeled at RTL or behavioral-level as *multivariate polynomials of finite degree* [Peymandoust and DeMicheli 2003; Smith and DeMicheli 2001]. In such designs, the bit-vectors have prespecified, finite word-lengths. For efficient and correct modeling of these designs, it is important to account for the effect of bit-vector size on the resulting computation. For example, the largest (unsigned) integer value that a bit-vector of size m represents is $2^m - 1$; implying that the bit-vector represents integer values reduced modulo 2^m (mod 2^m).

This suggests that bit-vector arithmetic can be efficiently modeled as algebra over finite integer rings, where the bit-vector size m dictates the cardinality of the ring Z_{2^m}. This article exploits the number-theoretic and algebraic properties of these rings to *engineer a high-level optimization technique* for lower-cost area implementation of polynomial datapaths.

Let us introduce the context in which the synthesis problems appear, and give a glimpse of the type and nature of these problems. Figure 1 depicts a typical design flow to realize arithmetic-intensive applications. Initial algorithmic specifications (such as a MATLAB or C model) of such systems represent the data in floating-point format. However, they are often implemented with fixed-point architectures (finite precision) in order to optimize the area-, delay-, and power-related costs of the implementation. Automatic conversion utilities are available for this purpose [Menard et al. 2002]. Subsequently, the fixed-point model is translated into an RTL description by using automatic conversion utilities, such as Groute and Keane [2000]. The resulting RTL is then synthesized into a circuit using high-level synthesis tools such as Synopsys [2007].

High-level synthesis has seen extensive research over the years. Various algorithmic techniques have been devised, and CAD tools developed, that are quite adept at capturing the hardware description language (HDL) models and mapping them into control/data-flow graphs (CDFGs), performing scheduling and resource allocation, sharing, binding, and retiming [DeMicheli 1994]. However, these tools lack the mathematical wherewithal to perform sophisticated algebraic manipulation for arithmetic datapath-intensive designs. While symbolic algebra-based manipulations have been used in high-level synthesis [Smith and DeMicheli 2001; Peymandoust and DeMicheli 2003; Verma and

Ienne 2006], they often neglect the effect of bit-vector size (m) on the computations while performing the optimization. This article demonstrates that polynomial manipulation while keeping the bit-vector size (m) in mind offers further potential for optimization. We develop an integrated approach to high-level synthesis that shows how bit-vector word-lengths can be used to devise new algebraic transformations for polynomial datapath optimization at RTL.

1.1 Motivation

Due to a large number of ADD, MULT operations in datapath designs, designers often employ ingenious strategies to control datapath size. In many cases, the design choice is that of a *single, uniform system word-length* for the computations [Kum and Sung 1998]. In such designs, m-bit adders and multipliers produce an m-bit output; only the lower m-bits of the outputs are used and the higher-order bits are ignored. Usually, such computations require appropriate scaling of coefficients and/or signals such that "overflow" can be avoided/ignored and standard fixed-point arithmetic can be implemented.

When the datapath size (m) over the entire design is kept constant, then fixed-size bit-vector arithmetic manifests itself as *polynomial algebra over finite integer rings* of residue classes Z_{2^m}; integer addition and multiplication is closed within the finite set of integers $0, \ldots, 2^m - 1$. Over such finite rings, symbolically distinct polynomials (those with different degrees and coefficients) can become computationally (bit-true) equivalent. This suggests that concepts from finite ring algebra can be exploited to *transform* a given polynomial computation into another, equivalent one for efficient hardware implementation.

Example 1 (*Arithmetic with Fixed-Size Bit-Vectors*). Consider the 5th-degree polynomial implementation of a polynomial filter used in image processing applications, shown in Eq. (1) [Jeraj 2005]. This filter is implemented as a uniform 16-bit RTL datapath.

$$\begin{aligned} F_1[15:0] = \; & 16384 * (X[15:0])^5 + 19666 * (X[15:0])^4 + \\ & 38886 * (X[15:0])^3 + 16667 * (X[15:0])^2 + \\ & 52202 * X[15:0] + 1 \end{aligned} \tag{1}$$

Another implementation of the same polynomial is given by

$$\begin{aligned} F_2[15:0] = \; & 3282 * (X[15:0])^4 + 22502 * (X[15:0])^3 + \\ & 283 * (X[15:0])^2 + 52202 * X[15:0] + 1. \end{aligned} \tag{2}$$

The polynomials F_1 and F_2 have different degrees and coefficients. However, because of fixed-size implementation, they are computationally equivalent. Mathematically speaking, $F_1 \neq F_2$ but it can be shown that $F_1[15:0] \equiv F_2[15:0]$ or $F_1 \mod 2^{16} \equiv F_2 \mod 2^{16}$. Moreover, F_1 requires 15 MULT and 5 ADD operations, whereas F_2 requires only 10 MULT and 4 ADD operations. When synthesized by the Synopsys module compiler [Synopsys 2007] using components from the DesignWare library, F_2 indeed occupies less area (28,840 sq.units) than F_1 (42,910 sq.units).

In the previous example, F_2 has a lower degree and fewer monomial terms than F_1. Intuitively, this "reduction" accounts for fewer ADD, MULT operations in F_2, and hence the lower implementation cost. So, given a fixed-bit-width (m) arithmetic computation $F[m-1:0]$, how can we derive a bit-true equivalent computation $G[m-1:0]$ with a lower implementation cost (if one exists)?

Example 2 (*Arithmetic with Composite Moduli*). The fixed-size datapath problem is somewhat restrictive. Often, datapaths contain multiple word-length operands. For example, consider the computation performed by a digital image rejection/separation unit that takes as two input signals: a 12-bit vector $A[11:0]$ and another 8-bit vector $B[7:0]$. These signals are outputs of a mixer wherein one signal emphasizes on the image signal and the other on the desired one. The design produces a 16-bit output Y_1. The computation performed by the design is described in RTL as shown in Eq. (3). Because of the specified bit-vector sizes, the computation can be equivalently implemented as another polynomial Y_2, as shown in Eq. (4).

input $A[11:0]$, $B[7:0]$;
output $Y_1[15:0]$, $Y_2[15:0]$;

$$\begin{aligned} Y_1 &= 16384 * (A^4 + B^4) + 64767 * (A^2 - B^2) + A \\ &\quad - B + 57344 * A * B * (A - B) \qquad (3) \\ Y_2 &= 24576 * A^2 * B + 15615 * A^2 + 8192 * A * B^2 \\ &\quad + 32768 * A * B + A + 17153 * B^2 + 65535 * B \qquad (4) \end{aligned}$$

Once again, the polynomials are symbolically distinct ($Y_1 \neq Y_2$), as they have different degrees and coefficients. However, because of the specified word lengths of the input and output operands, $Y_1[15:0] \equiv Y_2[15:0]$. Such arithmetic datapaths with multiple word-length architectures can be analyzed as polynomial functions from $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \cdots \times Z_{2^{n_d}}$ to Z_{2^m} [Chen 1996]. So, given an arithmetic computation with specified input/output bit-vector sizes, how do we derive an equivalent computation with a lower implementation cost?

This article addresses the preceding problems by integrating finite ring algebra and number theory within a CAD-based high-level synthesis framework.

1.2 Contributions

Contemporary symbolic algebra systems provide a variety of polynomial manipulation engines, such as decomposition, factorization, ideal membership testing (Gröbner's bases), etc., and have been employed for hardware optimization [Peymandoust and DeMicheli 2003; Smith and DeMicheli 2001]. However, such transformations are well defined over infinite fields such as reals (R), fractions (F), integral domains (Z), prime rings (Z_p), and/or Galois (finite) fields $GF(p^m)$; these are collectively called the *unique factorization domains* (UFDs). In contrast, the finite integer ring Z_{2^m} (formed by m-bit vectors) is a non-UFD because the bit-vectors lack multiplicative inverses and correspondingly, due

to the presence of zero divisors (e.g., $4 \neq 2 \neq 0$ but $4 \cdot 2 = 0 \bmod 8$). This *disallows* the use of some fundamental algebra techniques, such as Euclidean division and factorization [Allenby 1983] and Gröbner's bases (as applied in Peymandoust and DeMicheli [2003]).

On the other hand, commutative algebra and algebraic geometry offer a qualitative study of polynomials with coefficients in rings with many nilpotent elements (an element x of a ring is nilpotent if $x^n = 0$ for some positive integer n), such as Z_{2^m}. Therefore, we exploit some results from these areas [Singmaster 1974; Hungerbuhler and Specker 2006; Chen 1996, 1995] and apply them to the problem at hand. We study these problems from the very basic, fundamental perspective of polynomial functions over finite rings, particularly those of the type Z_{2^m}. We view non-UFD (Z_{2^m}) not as a liability, but as a resource that offers potential for optimization of bit-vector arithmetic.

Problem Modeling. We model arithmetic datapaths over fixed-size bit-vectors as *polynomial functions over finite rings* of residue classes Z_{2^m}. Let $x_1, x_2, \ldots, x_d$ denote the d-variables (bit-vectors) in the design. Let $n_1, n_2, \ldots, n_d$ denote the size of the corresponding bit-vectors. Therefore, $x_1 \in Z_{2^{n_1}}, x_2 \in Z_{2^{n_2}}, \ldots, x_d \in Z_{2^{n_d}}$. Specifically, Z_{2^n} corresponds to the finite set of integers $\{0, 1, \ldots, 2^n - 1\}$. Let m correspond to the size of the output bit-vector f, hence $f \in Z_{2^m}$. Subsequently, we model the arithmetic datapath computation as a *polynomial function* from $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \cdots \times Z_{2^{n_d}}$ to Z_{2^m} [Chen 1996]. Here $Z_a \times Z_b$ represents the *Cartesian product* of Z_a and Z_b. In other words, the computation is modeled as a multivariate polynomial $f(x_1, \ldots, x_d) \bmod 2^m$, where each $x_i \in Z_{2^{n_i}}$ and f is computed mod 2^m.

When $n_1 = n_2 \ldots . n_d = m$, that is, when all input and output bit-vectors are of the same size m, the polynomial function reduces to that of $f : Z_{2^m}[x_1, \ldots, x_d] \rightarrow Z_{2^m}$. Moreover, when $d = 1$, it becomes the case of a univariate polynomial function from $Z_{2^m} \rightarrow Z_{2^m}$. Typically, elementary functions such as $sin(x)$, $cos(x)$, $cot(x)$, etc., are implemented as univariate polynomials where x is the input bit-vector.

Approach. Using the concept of *polynomial reducibility* [Hungerbuhler and Specker 2006; Singmaster 1974; Chen 1996, 1995] over finite integer rings, we derive a step-by-step algorithmic procedure to reduce a given polynomial to a *unique, minimal, reduced* form. This reduced form is, in fact, a canonical representation of polynomial functions over finite integer rings. Moreover, we also derive an analytical model to estimate the implementation area of the datapath computation at polynomial level, by accounting for the degree of the polynomial (multipliers), the number of terms (additions), as well as the coefficients (logic simplification due to constant propagation). Subsequently, using this estimate along with the polynomial reduction procedure, we select the least-cost polynomial for implementation. Two algorithmic search procedures have been implemented for this purpose. Final synthesis results show average savings of around 23% over nonoptimized instances using our approach. Our finite-ring-algebra based approach fits seamlessly within contemporary

high-level synthesis methodology and provides an *extra degree of freedom* in datapath optimization.

Article Organization. The next section presents related work in this area. In Section 3, we briefly review some preliminaries and also introduce the basic concept for the polynomial optimization presented in this work. In Section 4, we present a systematic reduction procedure for polynomials implemented over $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \cdots Z_{2^{n_d}}$ to Z_{2^m}. Section 5 presents a cost model to estimate the implementation area at a polynomial level. In Section 6, we present an algorithm to further optimize the search procedure while performing the polynomial reduction. Section 7 presents the experimental results with an application and Section 8 concludes the article with some pointers to future work.

2. RELATED WORK

Contemporary high-level synthesis tools are quite adept in extracting control/data-flow graphs (CDFGs) from given RTL descriptions, and also in performing scheduling, resource-sharing, retiming, and control synthesis. However, they are limited in their capability to employ sophisticated algebraic manipulations to reduce the cost of the implementation [Peymandoust and DeMicheli 2003]. To overcome this limitation, there has been increasing interest in exploring the use of algebraic manipulation for RTL synthesis of arithmetic datapaths. The works of Smith and DeMicheli [1998] and Smith and DeMicheli [1999] derive new polynomial models of complex computational blocks for efficient synthesis. In Peymandoust and DeMicheli [2003], symbolic computer algebra tools are used to search for a decomposition of a given polynomial according to available library elements, using a Gröbner's bases-based approach. However, the derived polynomial models represent the computations over fields of reals (R), fractions (Q), or over the integral domain (Z), collectively called the unique factorization domains (UFDs). This often results in a polynomial approximation [Smith and DeMicheli 2001], without properly accounting for the effect of bit-vector size (m) on the resulting computation. Moreover, Buchberger's algorithm on Gröbner's bases, which has been used in Peymandoust and DeMicheli [2003], operates only on UFDs and cannot be directly ported over non-UFDs of the type Z_{2^m}. While the work of Constantinides et al. [2001] does account for datapath size for allocation, it operates directly on the original (given) arithmetic expression, thus limiting the degree of freedom in searching for a better implementation.

Other algebraic transforms have also been explored for efficient hardware synthesis: factorization and common subexpression elimination [Hosangadi et al. 2005, 2004] exploiting the structure of arithmetic circuits [Verma and Ienne 2004], term rewriting [Arvind and Shen 1998], etc. However, these techniques employ straightforward algebraic transforms and also overlook the effect of bit-vector size on the given computation.

In the area of logic optimization, various spectral transforms of Boolean functions have been derived for efficient synthesis of arithmetic circuits [Thornton et al. 2001]. Similar polynomial models for random logic circuits have also

been derived over Galois fields GF(2^m) [Pradhan 1978; Rajaprabhu et al. 2004; Pradhan et al. 2003], so that polynomial-algebra-based manipulation can be employed for logic optimization. While these works find application at the *circuit-netlist level*, they are not scalable enough to address polynomial bit-vector computations.

Note that our approach does not preclude some of the aforementioned synthesis procedures [Hosangadi et al. 2005, 2004; Constantinides et al. 2001]; it can be combined with these approaches as an additional optimization step. Modulo arithmetic has also been applied to the task of circuit/RTL verification [Huang and Cheng 2001]. The concept of polynomial functions over finite rings has also been applied to the equivalence verification of arithmetic datapaths in Shekhar et al. [2006, 2005]. This article demonstrates its application to *optimization* of arithmetic datapaths.

3. PRELIMINARIES

In this section, the material is mostly referred from Allenby [1983].

Definition 3.1. An *Abelian group* is a set G and a binary operation " $+$ " satisfying:

—Closure: For every $a, b \in G, a + b \in G$.
—Associativity: For every $a, b, c \in G, a + (b + c) = (a + b) + c$.
—Commutativity: For every $a, b \in G, a + b = b + a$.
—Identity: There is an identity element $0 \in G$ such that for all $a \in G, a + 0 = a$.
—Inverse: If $a \in G$, then there is an element $a^{-1} \in G$ such that $a + a^{-1} = 0$.

The set of integers Z, for instance, forms an Abelian group under addition.

Definition 3.2. A *commutative ring with unity* is a set R and two binary operations "$+$" and "$\cdot$" as well as two distinguished elements $0, 1 \in R$ such that R is an Abelian group with respect to addition with additive identity element 0, and the following properties are satisfied:

—Multiplicative closure: For every $a, b \in \mathrm{R}$, $a \cdot b \in \mathrm{R}$.
—Multiplicative associativity: For every $a, b, c \in \mathrm{R}$, $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.
—Multiplicative commutativity: For every $a, b \in \mathrm{R}$, $a \cdot b = b \cdot a$.
—Multiplicative identity: There is an identity element $1 \in \mathrm{R}$ such that for all $a \in \mathrm{R}$, $a \cdot 1 = a$.
—Distributivity: For every $a, b, c \in R$, $a \cdot (b + c) = a \cdot b + a \cdot c$ holds for all $a, b, c \in R$.

The set $Z_n = \{0, 1, \ldots, n-1\}$, where $n \in N$, also forms a commutative ring with unity. It is called the *residue class ring*, where addition and multiplication are defined *modulo* n (mod n) according to the rules to follow. For our application, $n = 2^m$.

$$(a+b) \bmod n = (a \bmod n + b \bmod n) \bmod n \tag{5}$$

$$(a \cdot b) \bmod n = (a \bmod n \cdot b \bmod n) \bmod n \tag{6}$$

$$(-a) \bmod n = (n - a \bmod n) \bmod n \tag{7}$$

Definition 3.3. Integers x, y are called *congruent modulo* n ($x \equiv y \mod n$) if n is a divisor of their difference: $n|(x - y)$.

Definition 3.4. A *zero divisor* is a nonzero element x of a ring R, for which $x \cdot y \equiv 0$, where y is some other nonzero element of R and the multiplication $x \cdot y$ is defined according to Eq. (6).

As an example, consider the nonzero integers 2 and 4 in the ring Z_8. Since $2 \cdot 4 \equiv 0 \mod 8$, 2 and 4 are zero divisors of each other. A commutative ring that has no zero divisors is known as an *integral domain*. The set of integers Z is an example.

Definition 3.5. A *field* F is a commutative ring with unity where every element in F, except 0, has a multiplicative inverse. In other words, for all $a \in F - \{0\}$, there exists an $\hat{a} \in F$ such that $a \cdot \hat{a} = 1$.

The system Z_n forms a field if and only if n is prime [Allenby 1983]. Hence Z_{2^m} (for $m > 1$) is not a field, as not every element in Z_{2^m} has an inverse. Lack of inverses in Z_{2^m} makes RTL verification complicated, since Euclidean algorithms for division and factorization are no longer applicable.

Definition 3.6. Let R be a ring. A *polynomial* over R in the indeterminate x is an expression of the form

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_k x^k = \sum_k a_i x^i \tag{8}$$

$\forall a_i \in R$. Elements a_i are coefficients, k is the degree. The element a_k is called the *leading coefficient*; when $a_k = 1$, the polynomial is monic.

The system consisting of the set of all polynomials in x over the ring R, with addition and multiplication defined accordingly, also forms a ring, called the *ring of polynomials* $R[x]$. Similarly, $R[x_1, \ldots x_d]$ denotes a ring of multivariate polynomials in d-variables. When $R = Z_{2^m}$, the corresponding polynomials are evaluated mod 2^m.

Early classical studies by Keller and Olson [1968], Kempner [1921], and Singmaster [1974] have analyzed functions over finite rings that have polynomial representations. These are generally termed as polynomial functions (or polyfunctions). Next, we state the definition given by Singmaster [1974].

Definition 3.7 [*Singmaster 1974*]. A function $f : Z_n \to Z_n$ is said to be a polynomial function if it is representable by a polynomial $F \in Z[x]$, that is, $f(a) \equiv F(a)$ for all a = 0,1, ... ,$n - 1$. Here, $\equiv$ denotes congruence mod n.

Example 3.1. Let $f : Z_3 \to Z_3$ be defined as $f(0) = 1$, $f(1) = 0$, $f(2) = 2$. It can be seen that f is a polynomial function, since f is representable by a polynomial $F(x) = 2x + 1$, namely, $f(a) \equiv F(a) \pmod 3$ for a = 0,1,2.

The preceding concept has been extended to multivariate polynomial functions, from $Z_n[x_1, x_2, \ldots, x_d] \rightarrow Z_n$ [Hungerbuhler and Specker 2006], where all variables and coefficients are in Z_n. This model is suited for arithmetic designs implemented with a fixed-size datapath, as they can be represented as polynomial functions over $Z_{2^m}[x_1, x_2, \ldots, x_d] \rightarrow Z_{2^m}$, where m is the bit-vector size of the computations.

This concept can be further extended to functions over $Z_{n_1} \times Z_{n_2} \times \cdots \times Z_{n_d} \rightarrow Z_m$. The following definition of such a polynomial function is taken from Chen [1996], and modified, for our application, to rings modulo an integer power of 2.

Definition 3.8. A function f from $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \cdots \times Z_{2^{n_d}} \rightarrow Z_{2^m}$ is said to be a polynomial function if it is represented by a polynomial $F \in Z[x_1, x_2, \ldots, x_d]$, that is, $f(x_1, x_2, \ldots, x_d) \equiv F(x_1, x_2, \ldots, x_d)$ for all $x_i \in Z_{2^{n_i}}$, $i = 1, 2, \ldots, d$ and $\equiv$ denotes congruence mod 2^m.

Example 3.2. Let $f : Z_{2^1} \times Z_{2^2} \rightarrow Z_{2^3}$ be a polyfunction in variables x, y, defined as: $f(0,0) = 1$, $f(0,1) = 3$, $f(0,2) = 5$, $f(0,3) = 7$, $f(1,0) = 1$, $f(1,1) = 4$, $f(1,2) = 1$, $f(1,3) = 0$. Then, f is a polyfunction representable by $F = 1 + 2y + xy^2$, since $f(x, y) \equiv F(x, y) \bmod 2^3$ for $x = 0, 1$ and $y = 0, 1, 2, 3$ (note that F is not a unique representation of f).

When $n_1 = \cdots n_d = m$, then this polyfunction reduces to $f : Z_{2^m}[x_1, \ldots, x_d] \rightarrow Z_{2^m}$, also represented as $f : Z_{2^m}^d \rightarrow Z_{2^m}$.

Definition 3.9. Let F, G be the polynomials $\in Z[x_1, x_2, \ldots, x_d]$ representing the functions f and g, respectively. We say that F is related to G, denoted as $F \equiv G$, if F and G induce the same polynomial function from $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \cdots \times Z_{2^{n_d}} \rightarrow Z_{2^m}$.

$F \equiv G$ implies that $F[x_1, x_2, \ldots, x_d] \equiv G[x_1, x_2, \ldots, x_d]$ for all $x_i = 0, 1, \ldots, n_i - 1$.

Example 3.3. Let $f : Z_8 \rightarrow Z_8$ be a polyfunction in x, defined as $f(0) = 0$, $f(1) = 4$, $f(2) = 4$, $f(3) = 0$, $f(4) = 0$, $f(5) = 4$, $f(6) = 4$, $f(7) = 0$. Then, f is a polyfunction representable by $F_1 = 6 * x^2 + 6 * x$ over $Z_8[x]$. Alternatively, f can also be represented as $F_2 = 2 * x^2 + 2 * x$ over $Z_8[x]$. Here, $F_1 \equiv F_2$ for all $x = 0, \ldots, 7$.

We wish to search for such equivalent polynomial representations in an efficient manner and identify a representation that is suitable for an optimized implementation. For this purpose, we introduce the concept of vanishing polynomials.

3.1 Vanishing Polynomials

In the previous example, $F_1 \equiv F_2 \bmod Z_8$. So, $F_1 - F_2 \equiv 0 \bmod 8$. Computing $F_1 - F_2$, we get $4 * x^2 + 4 * x$. For every value of $x \in Z_8$, $4 * x^2 + 4 * x$ computes to 0. Hence, in spite of being a polynomial with nonzero coefficients, $F_1 - F_2$ always computes to zero mod 8, or, in other words, $F_1 - F_2$ vanishes mod 8. Thus $F_1 - F_2$, represents a nil polyfunction. Such an expression is a *vanishing polynomial* over a finite ring.

A vanishing polynomial corresponds to *algebraic redundancy* in the computation. Eliminating such redundancies by subtracting "appropriate" vanishing polynomials from the given expression ultimately leads to its minimal, unique, canonical form. To create vanishing polynomials, we exploit some fundamental results in number theory.

Number theory perspective. According to a fundamental result in number theory, for any $n \in N$, $n!$ divides the product of n consecutive numbers. For example, 4! divides $4 \times 3 \times 2 \times 1$. But this is also true of any n consecutive numbers: 4! also divides $99 \times 100 \times 101 \times 102$. Consequently, it is possible to find the *least* $k \in N$ such that n divides $k!$ (denoted $n|k!$). We denote this value k as $k = SF(n)$, where $SF(n)$ is referred to as the *Smarandache function*.[1]

In the ring Z_{2^m}, let $SF(2^m) = k$ such that $2^m|k!$. For example, $SF(2^3) = 4$, since 8 divides $4! = 4 \times 3 \times 2 \times 1 = 2^3 \times 3$. But 8 does not divide 3!; hence least $k = 4$.

This property can be utilized to interpret the concept of vanishing polynomial as a divisibility issue in Z_{2^m}. If $f(x) \bmod 2^m \equiv 0$, then $2^m | f(x)$. In $Z_{2^3}[x]$, let $8|f(x)$. But $8|4!$ too, as $SF(8) = 4$. Therefore, if for all x, $f(x)$ can be represented as a product of 4 consecutive numbers, then $f(x)$ vanishes in Z_{2^3}. A polynomial can be represented as a product of 4 consecutive numbers as follows: $x(x-1)(x-2)(x-3)$.

Such a product expression is referred to as a *falling factorial* and is formally defined next.

Definition 3.10. Falling factorials of degree $k \in Z$ are defined according to $Y_0(x) = 1$, $Y_1(x) = x$, $Y_2(x) = x \cdot (x-1)$, $\ldots$, $Y_k(x) = x \cdot (x-1) \cdots (x-k+1)$.

Example 3.4. Consider $F(x)$ over $Z_{2^3}[x]$, where $F(x) = x^4 + 2x^3 + 3x^2 + 2x$. Here $SF(2^3) = 4$. $F(x)$ can be factored as a product of 4 consecutive numbers, namely, $(Y_4(x))$. Therefore $F(x)$ is a vanishing polynomial in $Z_{2^3}[x]$, or $F(x) \equiv Y_4(x) \bmod 2^3$, hence $F(x) \bmod 2^3 \equiv 0$.

The previous concept of falling factorials can be similarly defined for *multivariate expressions* over $Z_{2^m}[x_1, \ldots, x_d]$.

$$\boldsymbol{Y_k} = \prod_{i=1}^{d} Y_{k_i}(x_i) = Y_{k_1}(x_1) \cdot Y_{k_2}(x_2) \cdots Y_{k_d}(x_d) \tag{9}$$

Extending the aforesaid concept, if a multivariate polynomial in $Z_{2^m}[x_1, \ldots, x_d]$ can be factored into a product of $SF(2^m)$ consecutive numbers in *at least one of the variables* x_i, then it vanishes mod 2^m.

Example 3.5. Consider $F(x_1, x_2)$ over $Z_{2^2}[x_1, x_2]$, where $F(x_1, x_2) = x_1^4 x_2 + 2x_1^3 x_2 + 3x_1^2 x_2 + 2x_1 x_2$. Here $SF(2^2) = 4$, and the highest degrees of x_1 and x_2 are $k_1 = 4$ and $k_2 = 1$, respectively. F mod 4 can be equivalently written as $F = Y_{<4,1>}(x_1, x_2) \bmod 4 = Y_4(x_1) \cdot Y_1(x_2) \bmod 4$. Since F mod 4 can be represented as a product of 4 consecutive numbers in x_1, $2^2|F$ and $F \equiv 0$.

[1]This is a well-studied function in number theory. It was initially studied by Lucas [1883] and Kempner [1918] and recently revisited by Smarandache [1980].

When a polynomial cannot be factored into such Y_k expressions, can it still vanish? Consider the quadratic polynomial $4x^2 - 4x$ in $Z_8[x]$. It can be written as $4(x)(x-1)$. While $4x^2 - 4x$ cannot be factorized as $(x)(x-1)(x-2)(x-3)$ (a product of 4 consecutive numbers), it still vanishes in Z_8. The missing factors, $(x-2)(x-3)$ in this case, are replaced by the *multiplicative constant* 4; therefore $4x^2 - 4x \equiv 0 \bmod 8$.

In the next section, we explain how we identify appropriate vanishing polynomials and use them to devise a step-by-step polynomial reduction procedure in $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \cdots Z_{2^{n_d}}$ to Z_{2^m}.

4. POLYNOMIAL REDUCTION IN $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \cdots Z_{2^{n_d}}$ TO Z_{2^m}

In this section, we explain the polynomial reduction procedure in $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \ldots Z_{2^{n_d}}$ to Z_{2^m}. We use the following multiindex notation in the rest of the article [Hungerbuhler and Specker 2006]: $\boldsymbol{k} = \langle k_1, k_2, \ldots, k_d \rangle$ are the degrees corresponding to the d-variables $x = \langle x_1, x_2, \ldots, x_d \rangle$, respectively. We also provide lemmata and theorems that form the basis for the reduction procedure. Furthermore, we extend this technique to polynomial reduction of fixed-size datapaths (polynomial functions over Z_{2^m}) and explain it with suitable examples.

If a multivariate polynomial in $Z_{2^m}[x_1, \ldots, x_d]$ can be factored into a product of $SF(2^m)$ consecutive numbers in *at least one of the variables* x_i, then it vanishes mod 2^m (as explained in the earlier section). However, for a multivariate polynomial to vanish in $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \cdots Z_{2^{n_d}}$ to Z_{2^m}, we have to also take into account the input bit-vector sizes $(n_1, \ldots, n_d)$. For this purpose, we define a quantity.

$$\mu_i = \min\{2^{n_i}, SF(2^m)\}; i = 1, 2, \ldots, d \tag{10}$$

In the case of a fixed-size datapath, this equation reduces to

$$\mu_i = SF(2^m) = \mu; i = 1, 2, \ldots, d \tag{11}$$

because $n_1, n_2, \ldots, n_d = m$.

Now, consider the following:

LEMMA 4.1. *Let* $\boldsymbol{k} = \langle k_1, k_2, \ldots, k_d \rangle \in (Z^+)^d$, *where* Z^+ *represents the set of nonnegative integers. Then* $\boldsymbol{Y_k} \equiv 0$ *if and only if* $k_i \geq \mu_i$, *for some* i.

Example 4.1. Consider $F(x_1, x_2)$ over $Z_{2^1} \times Z_{2^2} \rightarrow Z_{2^3}$, where $F = x_1^2 x_2 - x_1 x_2$. Here $SF(2^3) = 4$, $k_1 = 2$, $k_2 = 1$. Furthermore, $\mu_1 = \min\{2^1, 4\} = 2 = k_1$, satisfying Lemma 4.1, and $\mu_2 = \min\{2^2, 4\} = 4 > k_2$. Now F can be written as

$$x_1^2 x_2 - x_1 x_2 \equiv x_1(x_1 - 1)x_2 \tag{12}$$

$$\equiv \boldsymbol{Y}_{<2,1>}(x_1, x_2) \tag{13}$$

$$\equiv 0. \tag{14}$$

The following example illustrates application of Lemma 4.1 for fixed-size datapaths.

Example 4.2. Consider $F(x_1, x_2)$ over Z_{2^3}, where $F(x_1, x_2) = x_1^4x_2 + 2x_1^3x_2 + 3x_1^2x_2 + 2x_1x_2$. Here $\mu = SF(2^3) = 4$ from Eq. (11), and $k_1 = 4$, $k_2 = 1$. Now $F(x_1, x_2)$ can be factored as $\boldsymbol{Y}_4(x_1)\boldsymbol{Y}_1(x_2)$, namely, $Y_{<4,1>}(x_1, x_2)$, where $\boldsymbol{Y}_4(x_1)$ is a product of 4 consecutive numbers, hence $\boldsymbol{Y}_4 \equiv 0 \mod 2^3$. Since the polynomial can be factored into a product of $SF(2^3)$ consecutive numbers in x_1, $F(x_1, x_2)$ is a vanishing polynomial in Z_{2^3}, or $F(x_1, x_2) \equiv \boldsymbol{Y}_{<4,1>}(x_1, x_2) \mod 2^3$, hence $F(x_1, x_2) \mod 2^3 \equiv 0$.

We now need to identify the constraints on multiplicative constants such that the given polynomial would vanish. We state the following result [Chen 1996].

LEMMA 4.2. *The expression* $g_{\mathbf{k}} \cdot \boldsymbol{Y}_{\boldsymbol{k}} \equiv 0$ *if and only if* $\frac{2^m}{\gcd(2^m, \prod_{i=1}^{d} k_i!)} | g_{\boldsymbol{k}}$, *where:*
—$g_{\mathbf{k}} \in Z$;
—$\mathbf{k} = \langle k_1, k_2, \ldots, k_d \rangle \in Z^d$ *such that* $k_i < \mu_i, \forall i = 1, \ldots, d$; *and*
—$gcd(2^m, \prod_{i=1}^{d} k_i!)$ *is the greatest common divisor of* 2^m *and* $\prod_{i=1}^{d} k_i!$
Henceforth, we will denote the term $\frac{2^m}{gcd(2^m, \prod_{i=1}^{d} k_i!)}$ *as* $b_{\boldsymbol{k}}$.

Example 4.3. Consider $F(x_1, x_2)$ over $Z_{2^1} \times Z_{2^2} \rightarrow Z_{2^3}$, where $F = 4x_1x_2^2 + 4x_1x_2$. Here $2^{n_1} = 2$, $2^{n_2} = 4$, and $2^m = 8$, $\boldsymbol{k} = \langle k_1, k_2 \rangle = \langle 1, 2 \rangle$. So $\prod_{i=1}^{2} k_i! = 1! \cdot 2! = 2$, $SF(2^m = 8) = 4$; $\mu_1 = \min\{2, 4\} = 2$, $\mu_2 = \min\{4, 4\} = 4$.

$$
\begin{aligned}
F &\equiv 4x_1x_2^2 + 4x_1x_2 && (15)\\
&\equiv 4 \cdot x_1 \cdot x_2 \cdot (x_2 - 1) && (16)\\
&\equiv g_{<1,2>} \cdot \boldsymbol{Y}_{<1,2>}(x_1, x_2) && (17)\\
&\equiv 0 && (18)
\end{aligned}
$$

because $b_{\boldsymbol{k}} = 4$, which divides $g_{<1,2>} = 4$.

In the following, we show an example illustrating application of Lemma 4.2 on fixed-size datapaths.

Example 4.4. Consider a polynomial $F(x_1, x_2)$ over Z_{2^3} where $F(x_1, x_2) = 4x_1^2x_2 - 4x_1x_2$. Here $g_{\boldsymbol{k}} = 4$, $\boldsymbol{Y}_{\boldsymbol{k}} = x_1(x_1 - 1)x_2 = \boldsymbol{Y}_{<2,1>}(x_1, x_2)$, and $k_1 = 2$, $k_2 = 1$. Computing $b_{\boldsymbol{k}}$, we get 4. Since 4 divides $g_{\boldsymbol{k}}$, $4x_1^2x_2 - 4x_1x_2$ is a vanishing polynomial.

The preceding concepts can be extended to derive a unique canonical representation for a polynomial function from $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \ldots Z_{2^{n_d}}$ to Z_{2^m}. We state the following theorem [Chen 1996].

THEOREM 1. *Let* F *be a polynomial representation for the function* f *from* $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \ldots Z_{2^{n_d}}$ *to* Z_{2^m}. *Then,* F *can be uniquely represented as*

$$F = \sum_{\boldsymbol{k}} c_{\boldsymbol{k}} \boldsymbol{Y}_{\boldsymbol{k}}, \tag{19}$$

where:

—$\boldsymbol{Y_k}$ *is the falling factorial defined in Eq.* (9);
—$\boldsymbol{k} = \langle k_1, \ldots, k_d \rangle$ *for each* $k_i = 0, 1, \ldots, \mu_i - 1$;
—$c_{\boldsymbol{k}} \in Z$ *such that* $0 \leq c_{\boldsymbol{k}} < b_{\boldsymbol{k}}$, *where* $b_{\boldsymbol{k}}$ *is as defined in Lemma* 4.2

PROOF. The proof is provided in Chen [1996]. Briefly reviewing it, any polynomial F from $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \ldots Z_{2^{n_d}}$ to Z_{2^m} can be decomposed in the form

$$F = \sum_{i=1}^{d} Q_i \, \boldsymbol{Y}_{\mu(i)} + \sum_{\boldsymbol{k}} a_{\boldsymbol{k}} b_{\boldsymbol{k}} \, \boldsymbol{Y}_{\boldsymbol{k}} + \sum_{\boldsymbol{k}} c_{\boldsymbol{k}} \, \boldsymbol{Y}_{\boldsymbol{k}}, \tag{20}$$

□

where:
—$Q_i \in Z[x_1, \ldots, x_d]$ is an arbitrary polynomial;
—$\mu(i) = <0, \ldots, \mu_i, \ldots, 0>$ is a d-tuple, where μ_i is in the position i and μ_i is defined according to Eq. (11);
— Y_k is the falling factorial defined in Eq. (9);
— $Y_{\mu(i)}$ is the falling factorial of degree $\mu(i)$ in variable x_i;
— $k = \langle k_1, \ldots, k_d \rangle$ for each $k_i = 0, 1, \ldots, \mu_i - 1$;
—$a_k \in Z$ is an arbitrary integer;
—b_k is defined according to Lemma 4.2; and
—$c_k \in Z$ is an arbitrary integer such that $0 \leq c_k < b_k$.

Let us consider the first term $(Q_i \, Y_{\mu(i)})$ from Eq. (20). This term is a vanishing polynomial according to Lemma 4.1, as $Y_{\mu(i)}$ is a falling factorial with degree μ_i. The second term $(\sum_k a_k b_k \, Y_k)$ is also a vanishing polynomial, as can be seen from Lemma 4.2. The third term $(\sum_k c_k \, Y_k)$ cannot be reduced any further, since the coefficient $c_k < b_k$ and hence b_k cannot divide c_k (for Lemma 4.2 to hold true). Hence, Eq. (20) can simply be written as $F = \sum_k c_k \, Y_k$.

The following example illustrates the aforesaid concept.

Example 4.5. Consider a polynomial $F = x_1^2 + 7x_1 + 6x_1x_2^2 + 2x_1x_2$ for $f: Z_2 \times Z_{2^2} \rightarrow Z_{2^3}$. Here $\mu_1 = \min\{2, SF(8)\} = 2$; $\mu_2 = \min\{2^2, SF(8)\} = 4$. Now F can be written as follows.

$$\begin{aligned} x_1^2 + 7x_1 + 6x_1x_2^2 + 2x_1x_2 &\equiv x_1(x_1 - 1) + 4x_1x_2(x_2 - 1) + 2x_1x_2(x_2 - 1) \\ &\equiv \boldsymbol{Y}_{<2,0>}(x_1, x_2) + a_{<1,2>} b_{<1,2>} \boldsymbol{Y}_{<1,2>}(x_1, x_2) \\ &\quad + c_{<1,2>} \boldsymbol{Y}_{<1,2>}(x_1, x_2) \\ &\equiv c_{<1,2>} \boldsymbol{Y}_{<1,2>}(x_1, x_2) \\ &\equiv 2x_1x_2^2 + 2x_1x_2 \end{aligned}$$

The first term $\boldsymbol{Y}_{<2,0>}(x_1, x_2)$ is a vanishing polynomial according to Lemma 4.1 ($Q_i = 1$). In the second term, $a_{<1,2>} = 1$, $b_{<1,2>} = 8/(8, 1! \cdot 2!) = 4$, thus making it a vanishing polynomial according to Lemma 4.2. In the third term, $c_{<1,2>} = 2$. It cannot be reduced any further, again according to Lemma 4.2. Thus, F can be written in the form given by Theorem 1, and is the unique canonical form representation of the polynomial.

The theorem can also be applied for the canonical form reduction of fixed-size datapaths (m), where $\mu_i = \mu = SF(2^m)\ \forall\ i$. This is illustrated in the following example.

Example 4.6. Consider a polynomial $F = x_1^4x_2 + 2x_1^3x_2 + x_1^2x_2 + x_1x_2$ for $f: Z_{2^2}^2 \rightarrow Z_{2^2}$. Here $k_1 = 4$, $k_2 = 1$ and $SF(2^2) = \mu = 4$. Specifical F can be written as follows.

$$\begin{aligned} x_1^4x_2 + 2x_1^3x_2 + x_1^2x_2 + x_1x_2 &\equiv x_1(x_1-1)(x_1-2)(x_1-3)x_2 + 2x_1(x_1-1)x_2 + x_1x_2 \\ &\equiv \boldsymbol{Y}_{<4,1>}(x_1,x_2) + a_2b_2\boldsymbol{Y}_{<2,1>}(x_1,x_2) + c_1\boldsymbol{Y}_{<1,1>}(x_1,x_2) \\ &\equiv \boldsymbol{Y}_1(x_2)\boldsymbol{Y}_{<4,0>}(x_1,x_2) + a_2b_2\boldsymbol{Y}_{<2,1>}(x_1,x_2) \\ &\quad + c_1\boldsymbol{Y}_{<1,1>}(x_1,x_2) \\ &\equiv c_1\boldsymbol{Y}_{<1,1>}(x_1,x_2) \\ &\equiv x_1x_2 \end{aligned}$$

The term $\boldsymbol{Y}_{<4,0>}(x_1,x_2)$ makes the first term a vanishing polynomial according to Lemma 4.1, where $Q_i = \boldsymbol{Y}_1(x_2)$. In the second term, $a_2 = 1$, $b_2 = 4/gcd(4,2!) = 2$. Hence, this term also vanishes according to Lemma 4.2. Finally, in the third term, $c_1 = 1$. Therefore, it cannot be reduced any further by using Lemma 4.2. Thus F can be written in the form given by Theorem 1, and is the unique canonical form representation of the polynomial.

4.1 Algorithm

Here, we present an algorithm that uses the concepts described earlier to reduce a polynomial function over $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \ldots Z_{2^{n_d}}$ to Z_{2^m}. The pseudocode for the algorithm is given in Algorithm 1, which takes the following inputs: Input polynomial F_1, d, variables $x_1, \ldots, x_d$, with corresponding input bit-widths $n_1, \ldots, n_d$ and the output bit-width m. The output of the algorithm is the polynomial in its unique canonical reduced form. The algorithm operates as follows:

Algorithm 1. RED_POLY: Reduce a Given Polynormal.

```
1: RED_POLY(F1, d, x, m, n).
2: F1 = Polynomial in x; d = Number of variables;
3: x[1...d] = List of input variables; m = Bit-width of F1;
4: n[1...d] = List of bit-widths of input variables, x;
5: poly = F1; Compute SF(2^m)
6: /*Compute the values for μi*/
7: for i = 1 to d do
8:    μ[i] = min(2^ni, SF(2^m)); k[i] = Max. degree of x[i] in poly;
9:    end for
10: /*Check if Y_μ(i) divides poly*/
11: for i = 1 to d do
12:    /*Lemma 4.1*/
13:    if (k[i] ≥ μ[i]) then
```

14: $quo, rem = \frac{poly}{Y_{<0,\ldots,k[i],\ldots,0>}(x_1,\ldots,x_d)}$;
15: **if** $(rem == 0)$ /* rem = remainder */ **then**
16: /*$poly = Q_\mu\ Y_{\mu(i)}$; a vanishing polynomial*/
17: return 0;
18: **else**
19: $poly = rem$;
20: *break*;
21: **end if**
22: **end if**
23: **end for**
24: /*Iterate over all possible degrees*/
25: **for** $j = \prod_{l=1}^{d}(\mu_l)$ to 1 **do**
26: /*Update degrees*/
27: **for** $i = 1$ to d **do**
28: $k[i]$ = Degree of $x[i]$ in the next highest order monomial of *poly*;
29: **end for**
30: $quo, rem = \frac{poly}{Y_{\langle k[0],\ldots,k[d]\rangle}(x_1,\ldots,x_d)}$;
31: $b_{\langle k[0],\ldots,k[d]\rangle} = \frac{2^m}{gcd(2^m, \prod_{i=1}^{d} k[i]!)}$;
32 : /*Lemma 4.2*/
33: **if** $(b_{\langle k[0],\ldots,k[d]\rangle} | quo)$ **then**
34: **if** $(rem == 0)$ **then**
35: return 0;
36: **else**
37: $poly = rem$;
38: **end if**
39: **end if**
40: c_k = Coefficient of $\langle k[0], \ldots, k[d]\rangle$
41: /*Check for the range of the coefficient*/
42: **if** $(c_k > b_{\langle k[0],\ldots,k[d]\rangle})$ /*if coefficient > the range*/ **then**
43: $quo, rem = \frac{poly}{b_{\langle k[0],\ldots,k[d]\rangle} * Y_{\langle k[0],\ldots,k[d]\rangle}(x_1,\ldots,x_d)}$;
44: $poly = rem$;
45: **end if**
46: Update *poly* w.r.t. its order;
47: **end for**
48: return *poly*;

---

(1) Assign F1 to poly.
(2) Compute $SF(2^m)$. A procedure to compute $SF(n)$ is outlined in the Appendix, which has a complexity of O(n/log n) [Power et al. 2002]. Here $n = 2^m$. The value of $SF(2^m)$ is then used to obtain the μ_i values.
(3) Find the max. degree (k_i) of each variable x_i in *poly*.
(4) Divide the polynomial by the falling factorial expressions $Y_{\mu(i)}$ in each of the d-variables.
(5) If the remainder is zero, it is a vanishing polynomial because $F = Q_\mu\ Y_{\mu(i)}$. Else, use the remainder as the new *poly*.

(6) Compute k_i (the degree of $x[i]$ in the next highest order monomial of the *poly*) and continue dividing from $Y_{\mu-1}$ (next highest degree) to Y_0 for each variable.

(7) After each division, check for the following conditions:
   —If the quotient can be written as $a_{\mathbf{k}} \cdot b_{\mathbf{k}}$ (where $b_{\mathbf{k}}$ is defined according to Theorem 1), and the remainder is zero, return 0. It is a vanishing polynomial.
   —If the quotient can be written as $a_{\mathbf{k}} \cdot b_{\mathbf{k}}$, and the remainder is nonzero, use the remainder as the new poly.
   —Check if the coefficient $c_{\mathbf{k}} > b_{\mathbf{k}}$. If so, perform the division with $b_{\mathbf{k}} * Y_{\mathbf{k}}$, and again use the remainder as the new poly.

Complexity. In Algorithm 1, the number of multivariate divisions is bound by $O(\prod_d \mu_i)$, where μ_i is as defined previously and d is the total number of variables.

Let us illustrate the operation of this algorithm in the following example.

Example 4.7. Consider a polynomial $F = x_1^2 + 7x_1 + 5x_1x_2^2 + 5x_1x_2$ over $Z_2 \times Z_{2^2} \rightarrow Z_{2^3}$.

—Initially the $SF(2^3)$ is computed ($SF(2^3) = 4$).
—The values of μ_i are then computed. $\mu_1 = 2$, $\mu_2 = 4$.
—The maximum degree of x_1 is found (k_1). Initially $k_1 = \mu_1 = 2$. Therefore Lemma 4.1 holds true. So we divide the *poly* F by $\mathbf{Y}_{<2,0>}(x_1, x_2)$.
—We get $quotient = 1$ and $remainder = 5x_1x_2^2 + 5x_1x_2$.
—Now $k_1 = 1$ and $k_2 = 2$. Since $\mu_2 = 4$, we come out of the loop.
—We divide the *poly* by $\mathbf{Y}_{<1,2>}(x_1, x_2)$. The $quotient$ is 5 and the $remainder$ is 0.
—Computing $b_{\langle k_1,k_2\rangle}$, we get 4. So $b_{\langle k_1,k_2\rangle}$ does not divide $quotient$.
—Now we divide the *poly* by $b_{\langle k_1,k_2\rangle} * \mathbf{Y}_{<1,2>}(x_1, x_2)$.
—Here $quotient = 1$ and $remainder = x_1x_2^2 + x_1x_2$.
—The $remainder$ is then assigned to the *poly*.
—$x_1x_2^2 + x_1x_2$ is the unique canonical reduced form for the given *poly*.

5. MODELING AREA COST AT POLYNOMIAL LEVEL

In Algorithm 1, at every reduction step we get an *intermediate* polynomial that is equivalent to the original. We wish to estimate the cost (implementation area) of the original polynomial, all intermediate polynomials, and also the final reduced form, and, then to select the least-cost expression for implementation. The minimal reduced form might not be the least expensive to implement; an intermediate expression might be less costly. This may happen when an intermediate form is "sparse" and the minimal form is more "dense."

Polynomial computations correspond to additions, multiplications, and constant multiplication operations (where one input to the multiplier is a constant). If we can compute the cost of the implementation area of these operations separately, we can determine the total cost of implementing any given polynomial f. Hence, we model the cost of adders, multipliers, and constant multipliers

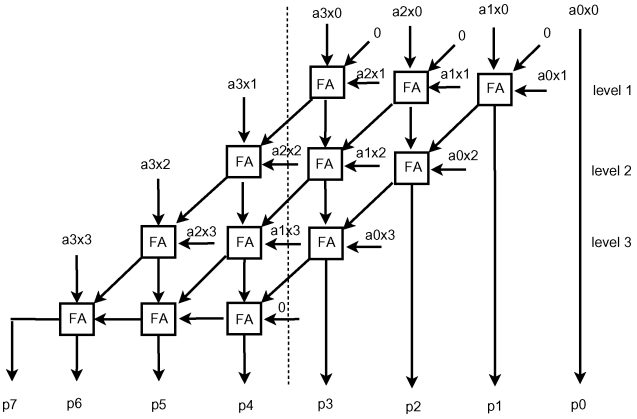

Fig. 2. Implementation of a 4-bit array multiplier (AX).

(implemented with specific input and output bit-vector sizes) at polynomial level.

—*Adders:* We estimate the area of an adder based on the implementation of a ripple-carry adder. If the input bit-vector sizes of the adder are n_1 and n_2, and the output bit-vector size is m:
—If $\max(n_1 + 1, n_2 + 1) > m$, then we require at least m full adder modules.
—else if $\max(n_1 + 1, n_2 + 1) < m$, then we will require $\max(n_1 + 1, n_2 + 1)$ full adder modules

$$Cost(Adder) = n * Cost(FA), \tag{21}$$

where $Cost(FA)$ is the cost of a full adder and n is the number of full adder modules.

—*Multipliers:* The estimated cost of an $n_1 \times n_2$ to m-bit multiplier is modeled on an array multiplier implementation [Koren 2002].

Consider the 4-bit array multiplier shown in Figure 2. It is composed of partial product generators and an array of full adder modules. Its area can be modeled as the sum of partial product cost and the array network cost. We are interested in the area of the multiplier responsible for generating only the lower m output bits. For instance, in Figure 2, if the value of m is 4, then the region of interest is to the right side of the dotted line. Therefore, the cost of the multiplier can be estimated as

$$Cost(mbit\_Mult) = Cost(PP(m)) + Cost(Arr(m)), \tag{22}$$

where $Cost(PP(m))$ is the cost of partial products (implemented with AND gates) and $Cost(Arr(m))$ is the cost of the array network (implemented with FA modules). Using the structure of the array multiplier and the values of n_1, n_2, and m, we can determine the minimum number of partial products and full adder modules required to implement an $n_1 \times n_2$ to m-bit multiplier.

—*Constant multipliers:* When an input to a multiplier is a constant, then the constant bits can be propagated to simplify the circuit. To model this effect,

we need to analyze their bit pattern and estimate a cost based on the simplification caused by propagating these bits. We model constant multiplication using the array multiplier model. An $n_1 \times n_2$ to m-bit constant multiplier is modeled as an $m \times m$ to m-bit multiplier (either by padding 0's or truncation) to apply our constant propagation strategy. In other words, if n_1 (or n_2) is smaller than m, then the remaining bits ($m - n_1$, (or $m - n_2$)) are padded with zeroes until the mth bit. On the other hand, if n_1 (or n_2) is greater than m, then only the lower-order m-bits (from n_1, and n_2) are chosen for the implementation. In this manner, for an $m \times m$ to m-bit multiplier, only the lower-order m-bits are analyzed for constant propagation.

Simplification using constant propagation. In Figure 2, consider X as the constant and A the variable. To propagate the constant X, we analyze the bits from the least significant position ($X[0]$) to the most significant ($X[m-1]$). Here are some results that we have derived to estimate the area as a result of constant propagation.

(1) *While traversing X from its LSB to MSB, until we reach a bit position whose value is* 1*, the cost of the implementation is zero due to zero propagation:* Consider the bit-pattern of $X = \{X[m-1], X[m-2], \ldots, X[i] = 1, 0, 0 \ldots, X[0] = 0\}$. Here, $X[i]$ is the least significant bit, with value 1. The partial products generated using $X[k], k < i$ will be 0. Therefore, up to the ith level, 0's are fed into the full adder modules, which results in their complete elimination (simplification) up to $(i-1)$ levels.

(2) *Until we reach the second bit position with value* 1 *in X while traversing from its LSB to MSB, the cost of the implementation is still zero:* Consider the bit-pattern of $X = \{X[m-1], \ldots, X[k] = 1, 0, 0 \ldots, X[i] = 1, 0, 0 \ldots, X[0] = 0\}$. Here, $X[i]$ is least bit position, with value 1 and $X[k]$ is the next least bit position, with value 1. We know that the area cost due to the bits from $X[0]$ to $X[i-1]$ is zero from the previous result. The partial products that are generated by $X[i]$ keep propagating until the kth level because: (a) There are no carry signals generated in the ith level; and (b) every subsequent level until $(k-1)$ performs an addition with 0 (partial products due to $X[i+1]$ to $X[k-1]$ are 0).

(3) *On encountering the second bit position with value* 1 *in the traversal of X from its LSB to MSB, the full adder modules in that level can be optimized to half adder modules:* Consider the bit-pattern used in the previous result. The partial products generated by $X[i]$ and $X[k]$ are added at the kth level. However, the carry signals feeding the full adder modules in the kth level are 0. Hence, these can be optimized to half adder modules.

(4) *For subsequent levels, if the value of $X[i]$ at any level is* 0*, then the full adders in that level reduce to half adders:* Since the partial products generated due to $X[i]'s = 0$ are also 0, the full adders being fed by these partial products are simplified to half adders.

Based on the bit-pattern of the constants, the previous models are employed to estimate the effect of constant propagation on the multiplier area.

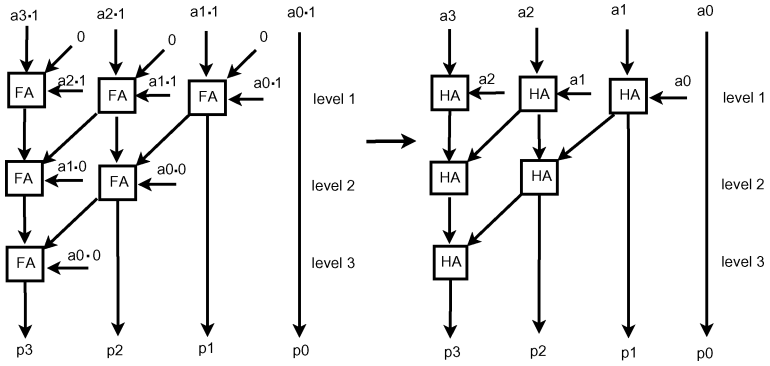

Fig. 3. Implementation of 3A, $X = 0011$.

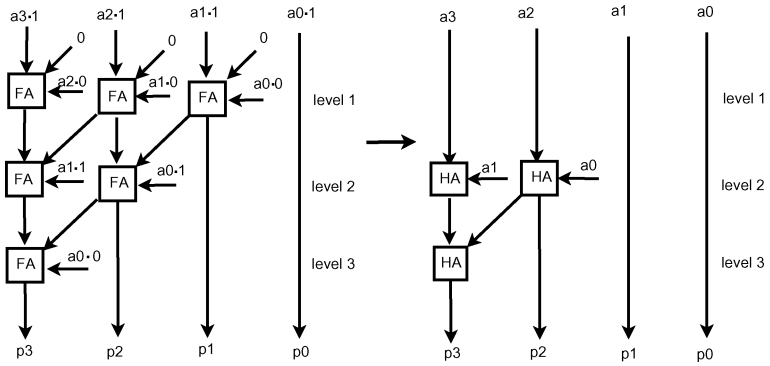

Fig. 4. Implementation of 5A, X = 0101.

Example. Consider the effect of $3 * A$ and $5 * A$ in a multiplier with output bit-vector size $m = 4$. Figures 3 and 4 depict the optimization in designs for the multiply operation with constants 3 (X = 0011) and 5 (X = 0101), respectively.

For $3 * A$ (constant $X = \{0011\}$), three full adders in level 1 are reduced to half adders, since one of the inputs to all the adders in this level is always zero. As the last two bits of X are 0, other full adders are also optimized to half adders. So the cost of $3 * A$ is $6 * Cost(HA)$.

For $5 * A$ ($X = \{0101\}$), the three full adders in level 1 are completely eliminated; two inputs to all the full adders in this level are zeroes. Hence, the results of the first level propagate to level 2; level-1 cost is zero. Moreover, full adders in levels 2 and 3 are reduced to half adders. Thus, the cost of $5 * A$ is equal to $3 * Cost(HA)$.

5.1 Quantifying the Cost

From the previous discussions, we see that the costs of all the modules are eventually expressed in terms of the costs of implementing AND gates, full adder, and half adder modules. We employ the unit model cost, that is, every logic gate can be implemented with a unit cost. A full adder can be optimally implemented with 5 2-input gates, while a half adder can be implemented using

2 2-input gates. We use these values as estimates to *quantitatively* calculate the area of the polynomial.

5.2 Integrated Approach

This cost model can now be integrated with the polynomial reduction Algorithm 1. In this algorithm, the *poly* gets updated in lines 19, 37, and 44. Every time the *poly* gets updated in these portions of the algorithm, we apply the cost model, estimate the cost at the polynomial level, and retain that polynomial with the least-cost implementation as the *min_cost poly*. On completion of the algorithm, *min_cost poly* gives the least-cost implementation of the polynomial.

6. BRANCHING ALGORITHM: POLYNOMIAL REDUCTION

The polynomial reduction technique presented in Algorithm 1 gives a systematic procedure to arrive at the unique canonical representation of the polynomial. By integrating it with the cost model, we can also find the polynomial with a lower-cost implementation. However, the reduction procedure can be further improved to search for other lower-cost polynomial implementations. Certain transformations that are left unexplored in the previous algorithm can be included in the search for a better implementation. In the following, we provide an example motivating the need for such an approach (presented in this section), and explain the necessary modifications to Algorithm 1 required for this approach.

Consider a polynomial $f = x^6 + 8x^3 + 8x$, with bit-vector sizes of {x, f} being $\{3, 4\}$, respectively. According to the previous algorithm, the reduction starts with the highest-degree monomial (*highest degree* $= 6$, in this case) and proceeds further. Using the previous algorithm, the polynomial reduction results in the following set of polynomials.

Initial polynomial: $f = x^6 + 8x^3 + 8x$;
1st intermediate polynomial: $f = 11 * x^5 + x^4 + 9 * x^3 + 8 * x^2 + 4 * x$;
2nd intermediate polynomial: $f = x^5 + 11 * x^4 + 7 * x^3 + 14 * x^2$; and
Final reduced polynomial: $f = x^5 + x^4 + 3 * x^3 + 12 * x$.

Using the cost model, the initial polynomial is estimated to be the least-cost polynomial, as it is much sparser than the others. However, in this polynomial, the subexpression $8x^3 + 8x$ is a vanishing polynomial in Z_{2^4}. Thus, it can be seen that if we choose to only reduce this subexpression, the initial polynomial f optimizes to x^6.

Initial polynomial: $f = x^6 + 8x^3 + 8x$
(*reduce only* $8x^3 + 8x$ and retain x^6 as is); thus
Optimized polynomial: $f = x^6$.

Now, the optimized polynomial has a lesser cost than the original. Thus, using an approach where subexpressions of the polynomial are selectively reduced, the optimization is further enhanced.

To lend an algorithmic procedure to such an approach, instead of iterating over all possible degrees (refer to Algorithm 1), we iterate over *all combinations* of all possible degrees. In other words, consider the previous example where $f = x^6 + 8x^3 + 8x$. The combination of all possible degrees is given by the set $\{(x^6+8x^3+8x), (x^6+8x^3), (8x^3+8x), (x^6+8x), (x^6), (8x^3), (8x)\}$. Each element of the set is considered as a subexpression, and it is reduced.[2] It should be noted that Algorithm 1 is subsumed in this algorithm. Since this is a more pervasive algorithm than the previous approach, the complexity clearly increases. In this algorithm, the number of multivariate divisions is bound by $O(\mu) = O(2^{\prod_d \mu_i})$ because in the worst case, it has to iterate through all combinations of all degrees for every variable to determine the optimized polynomial.

The pseudocode for this approach is given in Algorithm 2.

Algorithm 2. OPT_POLY: Optimize a Given Polynomial.

1: Given a polynomial F_1, compute the combination set of polynomials. $(F_{S_1}, F_{S_2}, \ldots . F_{S_{2^k-1}})$, where there are k monomial terms;
2: Initially, $min\_cost\_poly = F_1$; $min\_cost = Cost(F_1)$;
3: for every polynomial in the combination set, perform the reduction;
4: Call RED_POLY($F_{S_i}, d, x, m, n_{i's}$)
5: At every reduction step of F_{S_i}, determine the minimum cost polynomial ($min\_poly\_S_i$) with its corresponding cost ($Cost(min\_poly\_S_i)$);
6: /*At every reduction step, compute the cost of the entire polynomial (F_{new}) */ $Cost(F_{new}) = Cost(F_1 - F_{S_i}) + Cost(min\_poly\_S_i)$;
7: /* At every reduction step, retain the minimum cost polynomial. In other words, update min_cost_poly and min_cost */
if ($Cost(F_{new}) < min\_cost\_poly$){
$min\_cost\_poly = (F_{new})$;
$min\_cost = Cost(F_{new})$;
8: }
9: **end for**
10: min_cost_poly gives the minimum cost polynomial and min_cost gives its estimated cost;

7. EXPERIMENTS

The algorithms were implemented in Perl with calls to Maple [2007], along with the presented cost model for optimizing the given polynomial. The polynomial representing the datapath and operating bit-vector size ($input/output$ - $n_1, n_2, \ldots . n_d/m$) was given as the input to the tool. Step-by-step reductions of the given polynomial were performed using our algorithms until a minimal form was obtained. For the original, minimal, and every intermediate polynomial generated, the implementation cost was estimated. The polynomial with the least estimated cost was selected for implementation.

We used the Synopsys design compiler to generate the required $n_1 \times n_2$ to m-bit adders and multipliers. These units were used subsequently as functional

[2]Note that if there are k monomial terms, the combination set will have $2^k - 1$ elements.

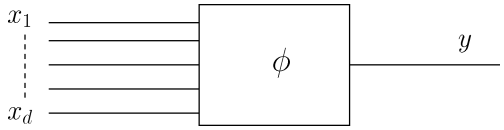

Fig. 5. A nonlinear filter model.

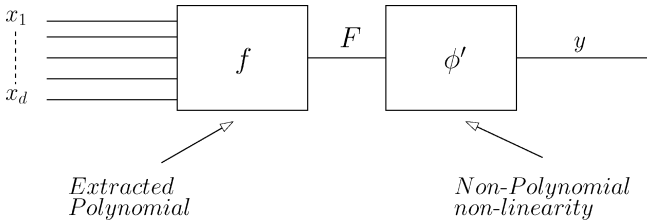

Fig. 6. Polynomial extracted model for the nonlinear filter.

units to implement the polynomials. To compare the area statistics, both the original and reduced polynomial with least estimated cost were implemented using the Synopsys module compiler.

7.1 An Application of our Approach

Let us demonstrate the application of our approach using a typical design methodology for nonlinear filters. Consider a nonlinear filter that needs to implement the function ϕ, as shown in Figure 5. Some nonlinearities can be approximated as polynomial functions. As shown in Figure 6, the polynomial function is extracted as f, while ϕ' is the nonlinearity of the system that cannot be expressed as a polynomial model. Generally, f is implemented using the Volterra series expansion model [Mathews and Sicuranza 2000]. Our focus essentially lies in optimizing f for a better implementation.

In Jeraj [2005], nonlinear systems were implemented using this design methodology. One example for such an extracted polynomial system model is the polynomial filter implemented in Jeraj [2005], used in an image processing application. Here, the polynomial representation is given by

$$F = a_1x^4 + a_2x^3 + a_3x^2. \tag{23}$$

By scaling the coefficients ($a_{i's}$) to an appropriate fixed-point representation, we determine their values and implement the polynomial over a uniform 16-bit datapath as

$$F = 13220 * x^4 + 16384 * x^3 + 8180 * x^2. \tag{24}$$

For the preceding RTL computation, when synthesized using the Synopsys module compiler [Synopsys 2007], using components from the DesignWare library, we get an implementation area of 25,384 sq.units.

On applying our optimization technique to this polynomial, we get a reduced expression for F as

$$F = 5028 * x^4 + 16372 * x^2 + 16384 * x. \tag{25}$$

The implementation area of this expression of F is only 19,904 sq.units.

The previous results suggest that for such applications, we can apply our optimization technique to effectively lower the final implementation area of the *extracted polynomial models*.

7.2 Results

Experiments have been performed on a variety of DSP benchmarks. The results for polynomial datapaths implemented with fixed-size bit-vectors are presented in Table I and those for polynomial datapaths implemented with multiple word-length operands are presented in Table II. In Table I, IRR is an image rejection receiver from Chen and Huang [2001]. Antialias is borrowed from Peymandoust and DeMicheli [2003]. Chebys (1)–(4) are Chebyshev polynomial filters from Hosangadi et al. [2004]. The last two functions are elementary function computations. In Table II, the first four examples are from Verma and Ienne [2004]. Deg4, Janez, and Cubic are polynomial filters used in image processing applications [Mathews and Sicuranza 2000]. Mibench is an automotive application from Guthaus et al. [2001]. PSK (phase shift keying) is from Peymandoust and DeMicheli [2003], and IIR-4 is a 4th-order IIR computation. For both the tables, column 2 lists the design characteristics: number of variables, their highest degree, and the bit-vector sizes ($n_1, \ldots n_d/m$). Column 3 lists the estimated cost of the original polynomial. Columns 4 and 5 list the cost of the optimized polynomial using Algorithm 1 and Algorithm 2, respectively. In columns 6 and 7, we list the percentage of improvement obtained in the estimated cost using Algorithm 1 (Imp1) and Algorithm 2 (Imp2), respectively. For the implementation cost, we report the results of Algorithm 2. Columns 8 and 10 list the actual implementation area of the original and selected polynomial (synthesized), respectively. Columns 9 and 11 depict the *critical path delay* of the original and selected polynomial implementations, respectively. These implementations have been realized using shifters, multipliers, and adders. Column 12 depicts the improvement in area of the actual implementation, while column 13 depicts the improvement in critical path delay in the implementation. The final column reports whether the polynomial chosen for implementation is the original polynomial (orig), the minimal canonical form (minimal), or a polynomial obtained during an intermediate reduction step (intermed). If the improvement in the estimated model is less than 1%, we choose the original polynomial for implementation.

For the benchmarks implemented with fixed-size bit-vectors, there is an average improvement of approximately 22.09%, with an improvement of 29.46% for the first 6 benchmarks from Table I. For the benchmarks implemented with multiple word-length operands, the average improvement in implementation area is 24.84% with an improvement of 35.49% considering only the first 7 benchmarks from Table II. Considering all the benchmarks, there is still an average improvement of 23.61% in the implementation area.

7.3 Consistency of our Estimation Approach

The cost estimated by our approach seems to be consistent with the actual implemented area of the designs. To elaborate further, we describe the

Table I. Performance Comparison of Estimation and Implementation Costs (fixed-size datapaths)

		Estimated Cost					Implementation Cost						Choice
Bench-mark	Var/Deg/m	Orig	Alg1	Alg2	Imp. 1	Imp. 2	Original		Alg2		Imp. %		
					%	%	Area	Delay	Area	Delay	Area	Delay	
IRR	2/4/16	10864	6943	6943	**36.09**	**36.09**	54594	400.04	37792	362.91	**30.77**	9.28	minimal
Antialias	1/7/16	18669	13972	13972	**25.15**	**25.15**	79254	540.12	59712	502.98	**24.65**	6.87	intermed
Cheby1	1/3/32	7521	6302	6302	**16.21**	**16.21**	42234	339.34	21490	251.96	**49.11**	25.34	minimal
Cheby2	1/4/32	15042	12604	12604	**16.21**	**16.21**	63724	503.92	42980	416.54	**32.55**	17.34	minimal
Cheby3	1/5/32	21184	18746	18746	**11.5**	**11.5**	94840	591.3	74096	503.92	**21.87**	11.56	minimal
Cheby4	1/6/32	26267	25048	25048	**4.67**	**4.67**	11630	755.88	95586	668.5	**17.83**	14.77	minimal
cot(x)	1/9/32	252658	252658	252658	<**1**	<**1**	—	—	—	—	—	—	orig
erf(x)	1/7/32	168190	168190	168190	<**1**	<**1**	—	—	—	—	—	—	orig

Table II. Performance Comparison of Estimation and Implementation Costs (arithmetic with composite moduli)

Benchmark	Var/Deg/	Estimated Cost				Implementation Cost						Choice
	$n_1,..n_d/m$	Orig	Alg1	Alg2	Imp.1	Imp.2	Original		Alg2		Improv. %		
					%	%	Area	Delay	Area	Delay	Area	Delay	
Poly1	3/4/14, 14, 16/16	7581	3927	3766	**48.2**	**50.3**	37430	372.45	20628	288.63	**44**	22.5	minimal
Poly2	3/4/10, 8, 13/16	4820	2393	2393	**50.3**	**50.3**	28848	288.63	11684	214.35	**59.49**	25.73	minimal
Poly3	2/5/13, 13/16	6227	5465	5465	**11.7**	**11.7**	28840	335.32	23006	298.18	**20.2**	11.07	minimal
Poly_unopt	1/4/12/16	5196	2994	2994	**42.3**	**42.3**	28836	335.32	14424	214.35	**49.9**	36.07	minimal
Deg4	3/4/16, 8, 16/16	22731	16361	16361	**28**	**28**	116684	632.44	82718	521.02	**29.1**	17.61	minimal
Janez	1/5/12/16	8907	6163	6154	**30.8**	**30.9**	42910	372.45	28840	335.32	**32.7**	9.97	minimal
Mibench	2/9/16,12/16	58510	48226	48226	**17.6**	**17.6**	249290	977.31	216772	921.07	**13.04**	5.75	intermed
PSK	2/4/11,14/16	18140	18140	18140	**<1**	**<1**	76876	—	—	—	—	—	orig
Cubic	3/3/24,28,31/32	47595	47586	47586	**<1**	**<1**	256388	—	—	—	—	—	orig
IIR-4	2/4/24,29/32	49339	49333	49333	**<1**	**<1**	213408	—	—	—	—	—	orig

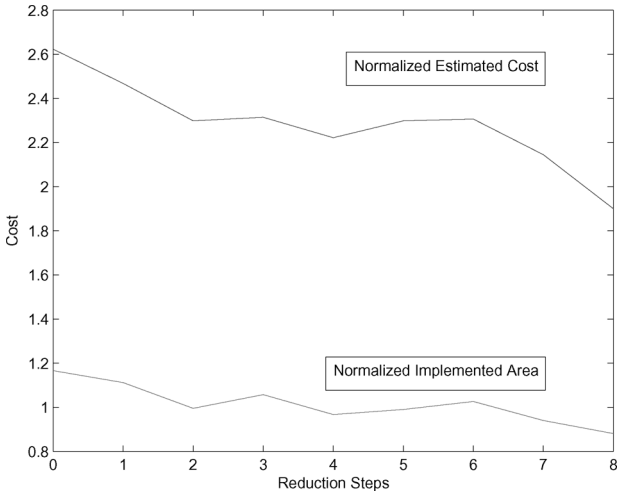

Fig. 7. Deg4: estimated and implemented costs for all reduction steps.

experiments performed with the Deg4 filter. Figure 7 plots the estimated cost and actual implemented area corresponding to the polynomials obtained after every reduction step. The figure shows that that the estimated cost behaves consistently with respect to the actual implementation area.

7.4 Horner Form Implementation

The Horner form of a polynomial is a nested normal form representation which expresses that polynomial with the minimal number of multiplications and additions. A Horner form for a univariate expression can be written as

$$a_0 \cdot x^n + \cdots + a_{n-1} \cdot x + a_n = (\cdots((a_0 \cdot x + a_1) \cdot x + a_2) \cdot x + \cdots a_{n-1}) \cdot x + a_n.$$

Multiply-accumulate units (MACs) are efficient hardware implementations of Horner polynomials. Typically, expressions are expressed in Horner form and directly mapped to MAC units. For the benchmark Poly3, we show how our cost model at the polynomial level is also suitable for their implementation on MAC units. Figure 8 plots the estimate, actual area of implementation using adders and multipliers, and the area of implementation using MAC units, and depicts this consistency.

7.5 Discussion on Expression Manipulation Techniques

There are many expression manipulation techniques that have been used in the optimization of arithmetic datapaths. While such techniques are commonly used in synthesis of arithmetic polynomials, our approach can be used as a *pre-processing step*, thus providing an additional scope for optimization. We briefly review conventional expression manipulation techniques [Peymandoust and DeMicheli 2003; Hosangadi et al. 2004] and contrast them against the optimization presented in our approach.

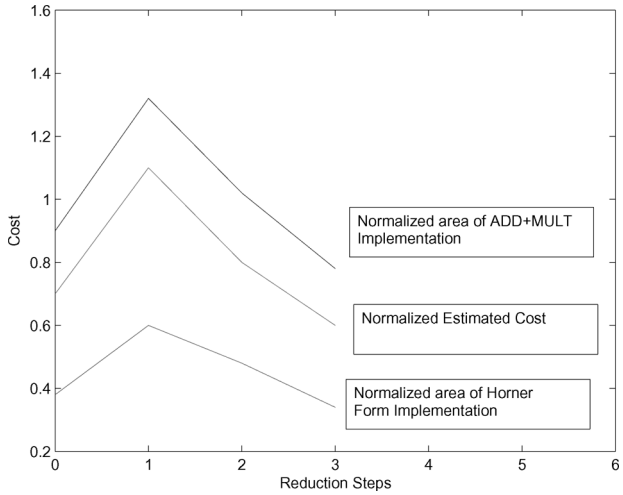

Fig. 8. Poly3: costs model versus add/mult and MAC implementation for all reduction steps.

—*Factorization:* Consider a polynomial $f = x^2 + 6 * x$. If f is a polynomial in Z (integral domain), then f can be uniquely factorized as $f = (x) * (x + 6)$ because Z is a unique factorization domain. However, if f is a polynomial in Z_{2^3} (fixed-size datapath with 3 bit-vectors), it can be factorized as $f = (x) * (x+6)$ or $f = (x+4)*(x+2)$ because Z_{2^3} is a nonunique factorization domain. (Note: $((x+4)*(x+2)) \bmod 2^3 = (x^2+6*x+8) \bmod 2^3 = (x^2+6*x) \bmod 2^3$ as $8 \bmod 2^3 = 0$). Unfortunately, factorization in non-UFDs is not a well-studied topic in symbolic algebra. For this reason, arithmetic datapaths implemented over fixed-size bit-vectors cannot fully exploit the benefit of factorization. In some sense, our approach exploits the nonunique factorization to search for better implementations.

—*Tree-height reduction:* In this technique, the focus is on reducing the height of an arithmetic expression tree, where the height of the tree is the number of steps required to compute the expression. For example, $(p + (q * r)) + (s)$ can be written as $(p + s) + (q * r)$.

—*Common subexpression elimination:* This is another expression manipulation technique, where isomorphic patterns are identified in the arithmetic expression tree and merged. This avoids the cost of implementing multiple copies of the same subexpression.

High-level synthesis techniques such as scheduling and resource sharing can also be employed to reduce the number of components and improve the critical path in an arithmetic expression.

For all the aforementioned techniques, it can be seen that they operate on the given data-flow graph (given computation) and will still need to implement all the operations shown in that graph. On the other hand, the data-flow graph generated by our approach leads to a better implementation. This graph can be further optimized by expression manipulation, scheduling, and resource-sharing techniques.

7.6 Limitation of our Approach

Given a polynomial f of degree $\mathbf{k}$, one can derive a vanishing polynomial q of higher degrees (say, $\mathbf{k}+\mathbf{1}$) as well. By computing $f+q$, one can create a higher-degree ($\mathbf{k}+\mathbf{1}$) polynomial equivalent to f. The cost of $f+q$ might be less expensive than f. Our approach cannot identify lower-cost implementations of a higher degree. Unfortunately, there can be more than one vanishing expressions of a given degree (depending upon the coefficients) that can be added to f. This makes it difficult to derive a "convergent" algorithm to search for low-cost implementations of higher degree.

8. CONCLUSIONS AND FUTURE WORK

This article has presented an area optimization approach for polynomial datapaths where the input and output bit-vector sizes of the operands are given as $(n_1, n_2, \ldots, n_d)$ and (m), respectively. Finite word-length bit-vector arithmetic is then modeled as a polynomial function from $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \cdots \times Z_{2^{n_d}}$ to Z_{2^m}. Exploiting the concept of vanishing polynomials over this mapping, we present two algorithms to optimize a given polynomial to a polynomial with a lower-cost implementation. A cost model to estimate the area at polynomial level is also presented. Using the optimization procedure along with the cost model allows to select an equivalent lower-cost expression for synthesis. Experimental results demonstrate substantial area savings over nonoptimized instances using our approach. Also, it can be seen that the area savings do not worsen timing. We are currently investigating how to extend our approach to perform polynomial decompositions over such arithmetic.

9. APPENDIX

Algorithm 3. $SF(2^m)$ Computation.

COMPUTE_SF(2^m)
2^m = Input for which *SF* value needs to be computed
/* For $m = 1$ *or* $m = 2$ */
if ($m \leq 2$)
  return $(2 \cdot m)$
end if
/* For $m > 2$ */
$v = \lfloor log_2(1+m) \rfloor$;
$rem = m$;
$n = 0$;
while ($rem \neq 0$) **do**
  $a_v = 2^v - 1$;
  $n = n + \lfloor rem/a_v \rfloor$
  $rem = rem \mod a_v$
  $v = v - 1$
end while
return $(m+n)$

$SF(n)$ was first considered by Lucas [1883], though the algorithm for its computation was outlined by Kempner [1918]. It was revisited in Smarandache [1980] and is defined as the smallest value for a given n at which $n|SF(n)!$. We have adapted the algorithm from Kempner [1918] for $n = 2^m$, and used the procedure as part of the algorithms in this article.

Example 9.1. Let us compute the value of $SF(2^3)$. In this case, $m = 3$. The algorithm proceeds as follows:

(1) Since $m > 2$, we compute the value of ν as $\lfloor log_2(1 + 3) \rfloor = 2$.
(2) Now, initialize $rem = 3$ and $n = 0$. Continue to the *while* loop since $rem > 0$.
    —Compute $a_2 = 2^\nu - 1 = 3$ and $n = 0 + \lfloor rem/a_2 \rfloor = 1$.
    —Update $rem = rem \bmod a_2 = 0$ and $\nu = \nu - 1 = 1$.
(3) Since $rem = 0$, exit the *while* loop. The computed value of $SF(2^3)$ is $(m+n) = 4$.

Complexity. The worst-case complexity of the algorithm is $O(m/log(m))$, where m corresponds to the word length of the output variable in the datapath.

REFERENCES

ALLENBY, R. J. B. T. 1983. *Rings, Fields, and Groups: An Introduction to Abstract Algebra.* E. J. Arnold.

ARVIND AND SHEN, X. 1998. Using term rewriting systems to design and verify processors. *IEEE Micro. 19*, 2, 36–46.

CHEN, C. AND HUANG, C. 2001. On the architecture and performance of a hybrid image rejection receiver. *IEEE J. Select. Areas Commun. 19*, 6, (Jun.) 1029–1040.

CHEN, Z. 1996. On polynomial functions from $Z_{n_1} \times Z_{n_2} \times \cdots \times Z_{n_r}$ to Z_m. *Discrete Math. 162*, 1–3, 67–76.

CHEN, Z. 1995. On polynomial functions from z_n to z_m. *Discrete Math. 137*, 1–3, 137–145.

CONSTANTINIDES, G., CHEUNG, P., AND LUK, W. 2001. Heuristic datapath allocation for multiple wordlength systems. In *Proceedings of the Design Automation and Test in Europe (DATE)* (Munich).

DEMICHELI, G. 1994. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York.

GROUTE, I. A. AND KEANE, K. 2000. M(VH)DL: A Matlab to VHDL conversion toolbox for digital control. In *IFAC Symposiun on Computer-Aided Control System Design* (Salford, UK).

GUTHAUS, M. R. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization* (Austin, TX).

HOSANGADI, A., FALLAH, F., AND KASTNER, R. 2005. Energy efficient hardware synthesis of polynomial expressions. In *Proceedings of the International Conference on VLSI Design* (Kolkata, India) 653–658.

HOSANGADI, A., FALLAH, F., AND KASTNER, R. 2004. Factoring and eliminating common subexpressions in polynomial expressions. In *Proceedings of the Internatoinal Conference on Computer Aided Design (ICCAD)* (San Jose, CA), 169–174.

HUANG, C.-Y. AND CHENG, K.-T. 2001. Using word-level atpg and modular arithmetic constraint solving techniques for assertion property checking. *IEEE Trans. Comput. Aided. Des. 20*, 381–391.

HUNGERBUHLER, N. AND SPECKER, E. 2006. A generalization of the smarandache function to several variables. *Integers: Electron. J. Combin. Num. Theory 6*, A23, 1–11.

JERAJ, J. 2005. Adaptive estimation and equalization of non-linear systems. Ph.D. thesis, University of Utah, Salt Lake City, Utah.

KELLER, G. AND OLSON, F. 1968. Counting polynomial functions (mod p^n). *Duke Math. J. 35*, 835–838.

KEMPNER, A. J. 1921. Polynomials and their residual systems. *Amer. Math. Soc. Trans. 22*, 240–288.

KEMPNER, A. J. 1918. Miscellanea. *Amer. Math. Month. 25*, 201–210.

KOREN, I. 2002. *Computer Arithmetic Algorithms*. A. K. Peters.

KUM, K. AND SUNG, W. 1998. Word-Length optimization for high-level synthesis of DSP systems. In *International Workshop Signal Processing Systems (SIPS)*. Piscataway, NJ.

LUCAS, E. 1883. Question nr. 288. *Mathesis 3*, 232.

MAPLE. 2007. Maple. http://www.maplesoft.com.

MATHEWS, V. J. AND SICURANZA, G. L. 2000. *Polynomial Signal Processing*. Wiley-Interscience, New York.

MENARD, D., CHILLET, D., CHAROT, F., AND SENTIEYS, O. 2002. Automatic floating-point to fixed-point conversion for DSP code generation. In *International Conference on Compiler, Architecture, and Synthesis Embedded Systems (CASES)* (Grenoble, France).

PEYMANDOUST, A. AND DEMICHELI, G. 2003. Application of symbolic computer algebra in high-level data-flow synthesis. *IEEE Trans. Comput. Aided. Des. 22*, 9, 1154–11656.

POWER, D., TABIRCA, S., AND TABIRCA, T. 2002. Java concurrent program for the smarandache function. *Smarandache Notions J. 13*, 1-2-3, 72–84.

PRADHAN, D. 1978. A theory of galois switching functions. *IEEE Trans. Comput. C-27*, 3, 239–248.

PRADHAN, D., ASKAR, S., AND CIESIESKI, M. 2003. Mathematical framework for representing discrete functions as word-level polynomials. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT)* (San Francisco, CA), 135–139.

RAJAPRABHU, T., SINGH, A., JABIR, A., AND PRADHAN, D. 2004. Modd for CF: A compact representation for multiple output function. In *IEEE International High Level Design Validation and Test Workshop* (Sonoma Valley, CA).

SHEKHAR, N., KALLA, P., AND ENESCU, F. 2006. Equivalence verification of arithmetic datapaths with multiple word-length operands. In *Proceedings of the Design Automation and Test in Europe (DATE)* (Munich).

SHEKHAR, N., KALLA, P., ENESCU, F., AND GOPALAKRISHNAN, S. 2005. Equivalence verification of polynomial datapaths with fixed-size bit-vectors using finite ring algebra. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)* (San Jose, CA).

SINGMASTER, D. 1974. On polynomial functions (mod m). *J. Num. Theory 6*, 345–352.

SMARANDACHE, F. 1980. A function in number theory. *Analele Univ. Timisoara, Fascicle 1*, 17, 79–88.

SMITH, J. AND DEMICHELI, G. 2001. Polynomial circuit models for component matching in high-level synthesis. *IEEE Trans. VLSI 9*, 6, 783–800.

SMITH, J. AND DEMICHELI, G. 1999. Polynomial methods for allocating complex components. In *Proceedings of the Design Automation and Test in Europe (DATE)* (Munich).

SMITH, J. AND DEMICHELI, G. 1998. Polynomial methods for component maching and verification. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)* (San Jose, CA).

SYNOPSYS. 2007. Synopsys module compiler and designware library. htpp://www.synopsys.com.

THORNTON, M., DRECHSLER, R., AND MILLER, D. M. 2001. *Spectral Techniques in VLSI CAD*. Kluwer Academic, Hingham, MA.

VERMA, A. K. AND IENNE, P. 2006. Towards the automatic exploration of arithmetic circuit architectures. In *Proceedings of the Design Automation Conference (DAC)* (San Francisco, CA).

VERMA, A. K. AND IENNE, P. 2004. Improved use of the carry-save representation for the synthesis of complex arithmetic circuits. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)* (San Jose, CA).

Received October 2006; revised March 2007; accepted March 2007