Title
Polynomial Datapaths Optimization

Permalink
https://escholarship.org/uc/item/86m4b4mx

Author
Parta, Hojat

Publication Date
2014

Peer reviewed|Thesis/dissertation
Polynomial Datapaths Optimization

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Hojat Parta

2014
The research presented focuses on optimization of polynomials using algebraic manipulations at the high level and digital arithmetic techniques at the implementation level. Previous methods lacked any algebraic understanding of the polynomials or only exposed limited potential. We have treated the polynomial optimization problem in abstract algebra allowing us algebraic freedom to transform polynomials. Unlike previous attempts where only a set of limited benchmarks have been used, we have focused on a class of polynomials called “Volterra series” with a wide range of applications in modeling nonlinear systems. We have exposed Volterra series characteristic, in which some inputs are delayed inputs, to store some of the products in these polynomials for the subsequent clock cycles. This results in replacement of multiplications with memory elements. In addition to variable factorization, we have also introduced factorization based on common coefficient divisors allowing us to expose more possible common subexpressions. These optimization have resulted in reduction of 41.0% and 12.9% in number of multiplication and addition respectively in comparison with common subexpression elimination technique used in compilers. At implementation level, instead of relying on conventional arithmetic operations, we have used redundant arithmetic to increase the speed of the polynomial evaluation. To avoid manual and tedious generation of such systems, we have developed a generator. This genera-
tor parses the polynomials into expression trees and automatically outputs Verilog HDL code. The generator uses interval arithmetic to find the minimum required bit-width in our evaluation of each variable. We have addressed the issues with truncation of redundant representation which unlike conventional arithmetic is not trivial. We have also used the non-uniform arrival time of operands in multi-operand addition to decrease the delay of adder trees. These improvements in implementation have resulted in 72.7\% increase in the speed of synthesized circuit in comparison with the implementation based on conventional operators. Finally we have used our methods on an application from digital communication domain. We have implemented a digital pre-distorter to reverse the effect of non-linearity in a satellite communication channel modeled using Volterra series. We have compared our results with previous attempts which relied on hand-optimization of similar systems. Our optimization techniques have resulted in an increase of 55.9\% in speed comparing to direct evaluation of Volterra series. When area optimization is the main objective, our approach has resulted in a decrease of 37.1\% and 73.0\% in number of DSP’s and Adaptive Logic Modules (ALM) respectively.
The dissertation of Hojat Parta is approved.

Majid Sarrafzadeh

Glenn Reinman

Sudhakar Pamarti

Miloš D. Ercegovac, Committee Chair

University of California, Los Angeles

2014
To my family
# Table of Contents

1 Introduction .................................................. 1

2 Polynomial Datapath Optimization .............................. 4
   2.1 Previous Work .............................................. 4
      2.1.1 Ideal Membership Approach ........................... 5
      2.1.2 Kernel/Co-Kernel Approach ............................ 6
      2.1.3 Vanishing Polynomials ................................. 7
      2.1.4 Partitioning and Compensation Heuristics .......... 9
      2.1.5 Coefficient Factorization and Square-Free Factorization .. 10
   2.2 Polynomial Optimization Formulation ...................... 11
      2.2.1 Subexpressions Extraction ............................ 12
      2.2.2 Subexpressions Compatibility Test ................... 13
      2.2.3 Optimization ........................................... 14
      2.2.4 Decomposition/Library Mapping ....................... 15
      2.2.5 Abstracted Algorithms ............................... 16
   2.3 Summary .................................................... 20

3 The Proposed Method ........................................... 22
   3.1 Possible Factorizations ................................ 23
   3.2 Subexpressions Compatibility Test ........................ 32
   3.3 Delayed Terms ............................................. 33
   3.4 Polynomials Rewriting ................................... 34
   3.5 Polynomial Optimization .................................. 35
3.5.1 Multiple-Term Optimization .................................... 35
3.5.2 Single-Term Optimization ...................................... 39
3.6 Results .................................................................. 45
3.6.1 Common Benchmarks ............................................ 48
3.7 Summary ................................................................ 49

4 Implementation .............................................................. 50
4.1 Redundant Digit Arithmetic ......................................... 50
  4.1.1 Multi-Operand Addition ........................................... 52
  4.1.2 Multiplications .................................................... 53
  4.1.3 Constant Multiplication ......................................... 65
  4.1.4 Truncation .......................................................... 67
4.2 Generator .................................................................. 70
  4.2.1 Parser ............................................................... 71
  4.2.2 Expression Tree Modification ................................... 73
  4.2.3 Size Calculator .................................................... 75
  4.2.4 Code Generator ................................................... 81
4.3 Results .................................................................. 87
4.4 Summary ................................................................ 90

5 Application ................................................................. 92
5.1 Volterra Series .......................................................... 92
  5.1.1 Linear Regression .................................................. 96
5.2 Power Amplifier Predistorter ....................................... 97
5.3 Results ................................................................ 103
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Sample Data-Flow Graph</td>
<td>46</td>
</tr>
<tr>
<td>4.1</td>
<td>[3:2] Adder</td>
<td>52</td>
</tr>
<tr>
<td>4.2</td>
<td>[4:2] Adder ($2\tau_{FA}$)</td>
<td>52</td>
</tr>
<tr>
<td>4.3</td>
<td>[4:2] Adder ($1.5\tau_{FA}$)</td>
<td>53</td>
</tr>
<tr>
<td>4.4</td>
<td>Multiplication $4 \times 4$ Signed Integers</td>
<td>55</td>
</tr>
<tr>
<td>4.5</td>
<td>Multiplication $4 \times 4$ Signed Integers - Simplified</td>
<td>56</td>
</tr>
<tr>
<td>4.6</td>
<td>2’s complement to minimally redundant Radix-4 Recoder</td>
<td>57</td>
</tr>
<tr>
<td>4.7</td>
<td>CS to Minimally redundant Radix-4 Recoder</td>
<td>58</td>
</tr>
<tr>
<td>4.8</td>
<td>CS bits partitioning</td>
<td>59</td>
</tr>
<tr>
<td>4.9</td>
<td>CS to Minimally redundant Radix-4 Recoder Implementation</td>
<td>61</td>
</tr>
<tr>
<td>4.10</td>
<td>CS to Minimally redundant Radix-4 Sign-Bit Recoder Implementation</td>
<td>63</td>
</tr>
<tr>
<td>4.11</td>
<td>Radix-4 Multiples Generator</td>
<td>65</td>
</tr>
<tr>
<td>4.12</td>
<td>Expression Trees</td>
<td>72</td>
</tr>
<tr>
<td>4.13</td>
<td>Fixed Expression Trees</td>
<td>73</td>
</tr>
<tr>
<td>5.1</td>
<td>Linearization using Predistorter</td>
<td>98</td>
</tr>
<tr>
<td>5.2</td>
<td>Nonlinear Amplifier</td>
<td>98</td>
</tr>
<tr>
<td>5.3</td>
<td>Nonlinear Communication</td>
<td>99</td>
</tr>
<tr>
<td>5.4</td>
<td>Predistored Nonlinear System Output</td>
<td>100</td>
</tr>
<tr>
<td>5.5</td>
<td>Predistortion Implementation</td>
<td>101</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Number of Arithmetic Operations</td>
<td>46</td>
</tr>
<tr>
<td>3.2</td>
<td>FPGA Implementation Costs</td>
<td>47</td>
</tr>
<tr>
<td>3.3</td>
<td>Number of Multiplications of Different Impl. of Predistorter</td>
<td>48</td>
</tr>
<tr>
<td>3.4</td>
<td>Benchmarks</td>
<td>48</td>
</tr>
<tr>
<td>4.1</td>
<td>Carry-Save (CS) Representation</td>
<td>51</td>
</tr>
<tr>
<td>4.2</td>
<td>Adder-Trees</td>
<td>54</td>
</tr>
<tr>
<td>4.3</td>
<td>Double Recoding</td>
<td>58</td>
</tr>
<tr>
<td>4.4</td>
<td>Sign Position Double Recoding</td>
<td>61</td>
</tr>
<tr>
<td>4.5</td>
<td>Generator Grammar</td>
<td>72</td>
</tr>
<tr>
<td>4.6</td>
<td>Variable Sizes and Ranges for ( { i = u_1 - u_2 + u_3; u_1 = a_2^2; u_2 = 5a_1a_2b_1; u_3 = #u_1; } )</td>
<td>81</td>
</tr>
<tr>
<td>4.7</td>
<td>Critical Path Variables Sizes</td>
<td>88</td>
</tr>
<tr>
<td>4.8</td>
<td>FPGA Implementation Costs</td>
<td>90</td>
</tr>
<tr>
<td>4.9</td>
<td>FPGA Implementation Maximum Frequency</td>
<td>90</td>
</tr>
<tr>
<td>5.1</td>
<td>Volterra Predistorter Model Parameters</td>
<td>101</td>
</tr>
<tr>
<td>5.2</td>
<td>Number of Arithmetic Operations</td>
<td>103</td>
</tr>
<tr>
<td>5.3</td>
<td>FPGA Implementation Costs</td>
<td>104</td>
</tr>
<tr>
<td>5.4</td>
<td>FPGA Implementation Maximum Frequency</td>
<td>105</td>
</tr>
<tr>
<td>5.5</td>
<td>Number of Multiplications of Different Impl. of Predistorter</td>
<td>105</td>
</tr>
<tr>
<td>A.1</td>
<td>List of functions used</td>
<td>116</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

It would never have been possible to take upon this journey without the people around me to support, guide, and inspire me.

I would like to thank my family and friends for their love and their belief in me. They have always given me the strength and support to continue and pursue my dreams.

I would like to express my gratitude to my Advisor, Dr. Miloš Ercegovac for his guidance and patience throughout these years. I’m grateful for his teachings, advice, and time. His constant advice of critically evaluating and questioning my work before others do, is one of the best lessons I take away.

I would also like to thank the committee members for their questions, feedback, and time.
VITA

Thesis: Realization of a Real-Time Reconfigurable Ultrasonic Target Detection System

2007-2010 MS Degree in Computer Science, UCLA.

Winter 2012 Teaching Assistant, CS111 Operating Systems Principles, Computer Science Department, UCLA.

PUBLICATIONS


CHAPTER 1

Introduction

Integer and floating-point datapaths implementing polynomials are widely found in practical applications such as digital signal processing (DSP) for multimedia applications [GK07, HFK06]. Arithmetic datapath-intensive designs implement a series of \texttt{add}, \texttt{mult} type computations over bit-vectors; therefore they are generally modeled at RTL or behavioral-level as \textit{systems of multivariate polynomials of finite degree} [PD03, SD01]. This has resulted in increasing interest in discovering algebraic manipulations and methods to optimize these polynomials. Several techniques such as Horner decomposition, factoring with common subexpression elimination [HFK06], common coefficient extraction [GK09] have been proposed. Symbolic computer algebra has also been employed to optimize datapath optimization by library component matching [PD03, SD01] and using vanishing polynomials over finite rings [GK07].

The presented research approaches the problem of polynomial datapaths optimization at two distinct level: At the high-level we have used algebraic methods to simplify the polynomials and at the implementation level we have utilized digital arithmetic techniques to optimize the hardware implementation. Our approach at the high-level is an extension of the Kernel/Co-Kernel [HFK06] by adding the factorization based on common coefficients’ divisors and including the optimization of products that can be buffered between clock cycles. We have treated the polynomial optimization problem in abstract algebra allowing us algebraic freedom to transform polynomials. Our approach generates set of all possible factorizations,
including the possible common coefficient factorization, of each polynomial in the set and then by formulating a minimum weighed rectangular covering problem, we find the best factorization which results in the highest reduction in the cost. In the next stage we find common products that can be extracted and used in different terms. We also find the products that can be buffered and used in the upcoming clock cycles, avoiding the calculation of the corresponding products in those clock cycles. These steps are repeated until we no longer can expose subexpressions to lower the cost of the implementation. At this stage we have optimized the set of polynomials and we pass the set of polynomials to the implementation stage. At implementation level, to avoid the time consuming effort of hand-coding the Hardware Description Language (HDL), we have implemented a generator. This generator applies digital arithmetic techniques such as redundant arithmetic, recoding and our own proposed techniques such as redundant representation truncation, redundant recoder, and arrival-time optimized adder trees to increase the speed of evaluation. The text is organized in following chapters:

- **Algebraic Polynomial Optimization** - Previous work in this area, motivation and formulation of the problem are presented.

- **Algebraic Stage** - Our algebraic method relying on factorization and common subexpression extraction by considering common coefficient factorization and delayed products are presented.

- **Implementation Stage** - Implementation details and techniques used to speed up evaluation are presented. Our generator to automatically generate HDL is presented as well.

- **Application** - The class of polynomials and an instance of non-linear systems modeled by Volterra series are presented. Results of the implementation of a digital predistorter to reverse the effect of non-linearity in a satellite communication is also presented.
• Appendix - The theoretical background, tools and additional information we have used throughout the text are presented. The HDL code for the building blocks we have implemented and used is also included.
CHAPTER 2

Polynomial Datapath Optimization

2.1 Previous Work

Factorization of polynomials have been studied in the literature [Len85, Gao03, Hoe02, Gua09], resulting in efficient algorithms widely used today. However focus of these methods are in factorizing a single polynomial whereas the objective in our research is to expose subexpressions that might not necessarily be factors of a single or a set of polynomials. This allows polynomials that are irreducible to still be decomposed to smaller blocks to be shared among a set of polynomials, reducing the number of multiplication or the overall cost of the implementation based on metrics such as area, propagation time or power. Fixed-size, bit-vector arithmetic manifest itself as polynomial ring over finite integer rings \((\mathbb{Z}_{2^m}[x_0, ..., x_n])\), which is a non-uniform factorization domain. This characteristic results in distinct polynomials (those with different degrees and/or different coefficients) becoming computationally equivalent (congruent modulo \(2^m\)). This has been used in [GK07] to optimize the implementation by minimizing the degree of the polynomial and forming a canonical form of the polynomial which cannot be further reduced. Common subexpression elimination (CSE) used in compilers [Muc97] is not suitable for optimizing polynomial expressions since they cannot perform factorization or extract common subexpression having more than two operands at a time [HFK06]. In [HFK06], each polynomial is factored to generate kernels/co-kernels representing each factorization. Finally the problem of finding the common subexpressions is formulated as a maximum weighed rectangular
covering problem defined over a matrix correlating the kernels/co-kernels with generated terms after factorizations. We have extended this method by using abstract algebra formulation and by considering different factorization possible for divisors of coefficients as well as considering products that are delayed version of products already calculated elsewhere in the polynomial set. In [PD01] Gröbner basis is used to optimize the synthesis of datapaths by matching the library components represented as polynomials. Ideal membership of the input polynomials in an Ideal, representing library components, is used for the mapping. However, this approach does not provide any means of automatically finding these library components and is only applicable to tuned library components. In [GKM07] linear blocks are extracted for polynomial decomposition however their effort is limited to simple blocks and common terms are not exposed in multi-polynomial sets. In [GK09] many of these techniques are combined together to optimize polynomials however we will discuss the limitations of these approaches. In following sections, we will introduce techniques proposed before to optimize polynomial datapaths. At the end of this chapter, the formulation of the problem will be presented as well.

2.1.1 Ideal Membership Approach

Ideal membership (Appendix A.1.2.1) checks if an element of a ring is a member of an ideal defined over that ring. The approach used in [PD01, PD03] uses ideal membership to map polynomials to library of hardware/software components represented as polynomials. This approach first creates an ideal of the library polynomials and calculates the Gröbner basis of that ideal. Checking membership of the polynomials in the datapaths upon this ideal, generated by Gröbner basis, will not only determine whether the polynomial is implementable but also it will give the coefficients of the linear combination of Gröbner basis used to implement the datapath polynomial. Major drawbacks of this method are its dependence
on available library elements and its sole applicability to optimization of a single polynomial. However our approach is applicable to the optimization of both individual and multiple polynomials and it automatically extracts and explores the design space without requiring a component library.

### 2.1.2 Kernel/Co-Kernel Approach

The approach used in [HKF05, HFK06, GK09] called “Kernel/Co-Kernel” has been the most successful optimization technique. This approach first find different ways of factoring a polynomial. The term used for factorization is called co-kernel and the quotient of factorization is called the kernel. For example polynomial $p = 4x + 4yz - xyz$ generates four sets of kernel/co-kernels: $(x + yz, 4)$, $(4 - x, yz)$, $(4 - yz, x)$, and $(4x + 4yz - xyz, 1)$. After all the kernels/co-kernels are generated the problem is formulated as a maximum weighed rectangular covering problem, solved using a greedy algorithm. The multiple-term common subexpressions are exposed by finding the intersection of kernels. The main drawback of this approach is the fact that it has been brought from logic synthesis [BM82], and it relies on factorization to find the common subexpressions without considering the possibility of common coefficient divisors. For example, for the set of polynomials $P = \{2x^2 + 6xy, 4xy^2 + 12y^3\}$ will result in following tuples of kernels/co-kernels:

$$
\{(2x + 6y, x), (2x^2 + 6xy, 1)\}
$$
$$
\{(4x + 12y, y^2), (4xy^2 + 12y^3, 1)\}
$$

We extend this method by considering coefficient divisors as well which would result in exposure of more common subexpressions allowing subexpression such as $x + 3y$ in the above example to be considered. We also extend this approach by considering products that can be delayed and used in next cycles due to the nature of Volterra polynomials where some of the variables are delayed values of...
other variables.

2.1.3 Vanishing Polynomials

The approach taken in [GK07] uses the fact that a fixed-size \( m \) bit-vector arithmetic manifests itself as \textit{algebra over finite integer rings} of residue classes \( \mathbb{Z}_{2^m} \). This results in symbolically different polynomials to become computationally equivalent under \( m \)-bit datapath assumption. The approach uses the fact from number theory that \( k! \) would divide any \( k \) consecutive numbers: \( k! \mid n(n - 1) \ldots (n - k + 1) \).

**Definition 1.** We define falling factorials of degree \( k \) as \( Y_k(x) = x(x - 1) \ldots (x - k + 1) \).

**Definition 2.** Smarandache function [Sma80], \( SF(n) \), defines the smallest \( k \) where \( n \mid k! \).

If we define \( k = SF(2^m) \) then any polynomial of the form \( x_i(x_i - 1) \ldots (x_i - k + 1) \) would vanish under \( m \)-bit datapath since:

\[
\begin{align*}
2^m & \mid k! \\
\; & \mid x_i(x_i - 1) \ldots (x_i - k + 1) \\
\therefore \\
& \equiv 0 \pmod{2^m}
\end{align*}
\]

The first statement comes from the definition of the Smarandache function. The second statement is from the number theory (\( k! \) divides any \( k \) consecutive numbers) and can easily be shown below:
\[
\binom{k+n}{k} = \frac{(k+n)!}{n!k!} = \frac{(n+k)(n+k-1)\cdots(n+1)}{k!}
\]
\[\therefore k! \mid (n+k)(n+k-1)\cdots(n+1)\]

However the paper presents the situation where the truncated falling factorial \((Y_{k'}(x)\text{ where } k' < k)\) form would also vanish if the coefficient has certain properties. This method is used in [AF10] using a canonical decision diagram called Horner Expansion Diagram (HED) for verification and optimization of polynomials. Similar method is used in [GK09] not for its property to vanish polynomials, but its ability to convert any polynomial to a canonical form which includes truncated falling factorials. Then each factor \((x-1), (x-2), \ldots, (x-k'+1)\) can be used to divide the set of polynomials and considered as common subexpression.

This has the limitation which will only expose limited expressions of the form \((x-1), (x-2), \ldots, (x-k'+1)\). The other limitations are the dependency of this method on the bit-size; as the bit-size increases the degree of the vanishing falling factorial polynomial increases. This approach exposes the fact that the computation bit-width is not large enough and some of the calculation is ignored due to modular arithmetic. Hence, the evaluated polynomial is different from the input polynomial. For example, we know \(SF(2^3) = 4\) and for the 3-bit datapath and the polynomial \(f(x) = x^4 + 6x^3 + 3x^2 + 6x\) defined over \(\mathbb{Z}_8[x]\), we have:

\[
f(x) = x^4 + 6x^3 + 3x^2 + 6x
\]
\[
= x(x+1)(x+2)(x+3)
\]
\[
\equiv 0 \pmod{2^3}
\]

So this polynomial is evaluated as 0 for the bit-width of 3. However, this is not
acceptable in practical applications and this issue is addressed by increasing the bit-width or using saturation arithmetic [EL03].

2.1.4 Partitioning and Compensation Heuristics

The approach used in [SAF09] and improved version in [AF09] are based on a heuristic that partitions the polynomial into a form $p_1 \times p_2 + p_3$. This approach starts by factoring the polynomial on a single variable. If there exists more than one monomial that contains this variable, the greatest common divisor (GCD) is entered into $p_1$ and the factored monomials into $p_2$. The algorithm continues by trying to partition on other variables if only adding to $p_1$ and $p_2$ doesn’t increase the cost of the implementation by adding new monomials to the polynomial. Anything that is not partitioned are left in $p_3$. The next step of the process compensate for the coefficients of these monomials to match the original polynomial. This can be done by solving a set of nonlinear integer equations obtained from $poly - p_3 = p_1 \times p_2$. The enhanced algorithm [AF09] improves on this algorithm by adding hidden monomials in the compensation step (monomial compensation). These are monomials that would allow a better partitioning which will reduce the overall cost of implementation. The issue with this approach is that as we increase the number of variables this becomes similar to factorization technique in Kernel/Co-Kernel and the coefficient compensation step become intractable. This method in addition to the method from section 2.1.3 (vanishing polynomials) are used in [AF10] to optimize polynomials. As we discussed most of the benefits (cost reduction) comes from vanishing polynomial method which disappears when the bit-width is increased to the required size.
2.1.5 Coefficient Factorization and Square-Free Factorization

In [GK09] a combination of different methods including the Vanishing polynomials, coefficient factorization and square-free factorization are used. As mentioned before the major reduction in cost is due to loss of computation by using small bit-width which in any practical application is not tolerable. The coefficient factorization proposed in [GK09] also has drawbacks. The common coefficient factorization is a separate step from the subexpression extraction and coefficient factorization is only considered for the case where a coefficient is multiple of another coefficient. For example for the polynomial $8x + 12y$, no coefficient factorization is considered, where a factorization of the form $4(2x + 3y)$ can expose $2x + 3y$ as a potential common subexpression. As will be explained we have integrated the coefficient factorization into our optimization step resulting in exposure of more subexpressions. The square-free factorization relies on the fact that if a polynomial is of the form $f(x) = c_i(x)^{d_i}c_{rest}(x)$ the derivative of the polynomial is:

$$f'(x) = d_i c_i(x)^{d_i-1}c_{rest}(x) + c'_{rest}(x)c_i(x)^{d_i}$$

And if we calculate the greatest common divisor of $f$ and its derivative we have:

$$\gcd(f(x), f'(x)) = c_i(x)^{d_i-1}$$

This can be extended to multivariate polynomials and by calculating derivatives, one can find the factors with powers more than 1 ($d_i > 1$). This is a simple step and has a polynomial time complexity. However we found that none of the polynomials we used had factors of this form and this is due to the fact that polynomials defined over integers are almost all irreducible [PL04], so they cannot be factored let alone have factors with power of 2 or higher.
2.2 Polynomial Optimization Formulation

Polynomial datapath of bit-width $m$ manifests itself as the *polynomial ring over finite integer ring* ($\mathbb{Z}_{2^m}[x_1, \ldots, x_n]$). Reduction in the number of operations in the evaluation of these polynomials would result in optimization of power, implementation cost, and propagation delay. Compiler techniques used to reduce these operations, lack the algebraic understanding of this problem and they are not as effective. We consider the polynomials defined over integers ($\mathbb{Z}[x_1, \ldots, x_n]$) since we do not rely on the modular nature of fixed datapath size. This allows us to apply our algebraic techniques regardless of the datapath size. We formulate the polynomial(s) optimization where previous methods can be mapped to all or portion of this model. The common steps in polynomial(s) optimization include:

- **Subexpressions Extraction** - The most important step of polynomial(s) optimization is the extraction of useful subexpressions for the decomposition of the polynomial(s). Based on the objective of the optimization, they can be common subexpressions to decrease overall implementation cost, or factors of a single polynomial allowing parallel implementation resulting in a faster circuits, or in certain architectures allowing decomposition of the polynomial into already available instructions/accelerators.

- **Subexpressions Compatibility Test** - In certain cases before we can optimize the design, we have to generate a compatibility graph of these subexpressions exposed in the previous step. Only then we can start the decision of what subexpressions would result in an optimized implementation.

- **Optimization** - The goal of this step is to optimize the implementation of the polynomial(s) based on a cost function. The cost function allows the application of these optimization methods to a wide range of applications.

- **Decomposition** - After finding the most suitable subexpressions in previous
steps, we would need to decompose the original polynomial(s) using these 
subexpressions (polynomials) which can be thought as library components 
automatically extracted from the polynomial(s).

In some approaches the first two steps are combined. For example the Kernel/Co-
Kernel method or our proposed method combine these two steps. In addition, 
these steps can be applied in iterations with different subexpression extractions 
methods at each iteration.

2.2.1 Subexpressions Extraction

This step is the most important portion of polynomial(s) optimization since the 
decomposition of the polynomial(s) relies directly on the results of this step. This 
is a very familiar concept in logic synthesis and mapping. Already built compo-
nents that are highly optimized are available at time of logic optimization. Similar 
methodology can be applied to arithmetic and in this case polynomial(s) optimiza-
tion. Ideal membership approach [PD01, PD03] relies on pre-built components. 
However manually designing these components is not always feasible or even pos-
sible. To allow automatic extraction of these components a lot of research has 
already gone into other areas of computer science such as automatic Instruction 
Set extraction [GB08, CZM05] and Arithmetic Exploration [VI06]. The other ap-
proaches in polynomial(s) optimization mentioned before rely on this step with 
limitations as explained before. This step, given the input polynomial(s), generate 
a set of all subexpressions \( \{c_1, c_2, \ldots, c_m\} \in \mathbb{Z}[x_1, x_2, \ldots, x_k] \) that can decompose 
the polynomial(s) in the final implementation. For example, for the set of poly-
nomial \( P = \{x^2 + 6xy + 9y^2 + yz + z^2, 4xy^2 + 12y^3 + x^2z + z^3\} \) we have following
potential subexpressions for each polynomial:

\[
x^2 + 6xy + 9y^2 + yz + z^2 \rightarrow \{6x + 9y + z, y + z, 2x + 3y, x + 6y\}
\]

\[
4xy^2 + 12y^3 + x^2z + z^3 \rightarrow \{x^2 + z^2, x + 3y, xz + 4y^2\}
\]

### 2.2.2 Subexpressions Compatibility Test

Subexpressions extracted in the last step are not all compatible; using a subexpression in the implementation can invalidate some of the other subexpressions for the implementation. Unlike previous attempts to formulate subexpression compatibility, we use polynomial division to formulate the compatibility function. The \texttt{quo} and \texttt{rem} are the quotient and remainder defined based on monomial-ordering insensitive division over term defined in section 3 or multivariate polynomial division defined in appendix. All the variables \(p, c, r, \ldots\) are polynomials \(\in \mathbb{Z}[x_1, x_2, \ldots, x_k]:\)

\[
if \ p = c \times q + r
\]

\[
q = \texttt{quo}(p, c)
\]

\[
r = \texttt{rem}(p, c)
\]

\[
\texttt{compatible}(p, c_1, c_2) = \begin{cases} 
  \text{True,} & \text{if } \texttt{quo}(\texttt{quo}(p, c_1), c_2) > 0 \lor \\
  \text{quo}(\texttt{rem}(p, c_1), c_2) > 0 \lor \\
  \texttt{quo}(c_1, c_2) > 0 \\
  \text{False,} & \text{otherwise} 
\end{cases}
\]

The subexpressions \(c_2\) is compatible with \(c_1\) if after performing the polynomial division of \(p\) over \(c_1\) we can still perform polynomial division on the quotient or remainder of the first division as well as if \(c_1\) can be decomposed by \(c_2\). The dependency graph generated with this definition is an undirected graph for mono-
mial ordering insensitive division and a directed graph for multivariate polynomial division. For example, for the polynomial \( f(x) = x^2 + 6xy + 9y^2 + yz + z^2 \) and the potential subexpressions \{6x + 9y + z, y + z, 2x + 3y, x + 6y\}, the following statements hold according to the polynomial division over term (section 3):

\[
\begin{align*}
\text{compatible}(f, 6x + 9y + z, y + z) &= False \\
\text{compatible}(f, 6x + 9y + z, 2x + 3y) &= False \\
\text{compatible}(f, y + z, 2x + 3y) &= True \\
\text{compatible}(f, y + z, x + 6y) &= True
\end{align*}
\]

**Definition 3.** We define a valid decomposition sequence of the polynomial \( f_i \) as an ordered set \((a_1, a_2, \ldots, a_q)\) where \( a_j \in [1, m] \) and:

\[
\begin{align*}
\exists p &\in \{f_i\} \cup \{\text{quo}(f_i, c_{a_1}), \text{rem}(f_i, c_{a_1})\} \\
&\cup \{\text{quo}(\text{quo}(f_i, c_{a_1}), c_{a_2}), \text{quo}(\text{rem}(f_i, c_{a_1}), c_{a_2})\} \\
&\cup \{\text{rem}(\text{quo}(f_i, c_{a_1}), c_{a_2}), \text{rem}(\text{rem}(f_i, c_{a_1}), c_{a_2})\}, \ldots\} \\
\text{compatible}(p, c_{a_j}, c_{a_{j+1}}) &= True
\end{align*}
\]

In this definition at each step we look for a possible subexpression \( p \) among \( f_i \) and all the expressions generated by the polynomial division in all the previous steps (quotients and remainders).

### 2.2.3 Optimization

The overall optimization problem can be formulated as following. The set of polynomials in the datapath is defined as \( f_1, f_2, \ldots, f_n \in \mathbb{Z}[x_1, x_2, \ldots, x_k] \). We also have the set of subexpressions from the first step as \( C = \{c_1, c_2, \ldots, c_m\} \subset \mathbb{Z}[x_1, x_2, \ldots, x_k] \). A valid decomposition sequence (Definition 3) is defined as ordered set \( A_i = (a_1, a_2, \ldots, a_q) \) is used to optimize the overall implementation.
of the input polynomial(s). We denote the decomposition of the polynomial \( f_i \) using polynomial division/factorization and a valid decomposition sequence as \( f_i \xrightarrow{A_i} \). The function \( \text{COST} \) defines the cost of polynomial(s) which can be customized depending on the overall objective and it does take into consideration the common subexpressions. The optimization can be formulated as:

\[
\text{argmin}_A \sum_{i=1}^{n} \text{COST}(f_i \xrightarrow{A_i})
\]

where \( A = \{ A_1, A_2, \ldots, A_n \} \) is the set of all valid decomposition sequences of the input polynomial(s). We should note that since optimization is on all the sequences for all the input polynomials, in the case of Multi Input Multi Output (MIMO) circuitry, some of these subexpressions will be the common subexpressions similar to [HKF05, HFK06, GK09].

### 2.2.4 Decomposition/Library Mapping

Once we have gone through the previous steps, we are left with an ordered set of most suitable subexpressions \( A = (c_1, c_2, \ldots, c_l) \) which is the subexpressions we need to decompose our set of polynomials with. This set can be regarded as the library components similar to when a designer extract them tediously by hand. To decompose the input polynomial(s), map these library components to the design, we perform division of polynomials (factor or multivariate division) over each extracted subexpression in order defined in the valid decomposition sequence. In the case that division is defined as multivariate polynomial division, using the polynomials from the optimization step as the generating polynomials, one may create an Ideal over our Polynomial ring which all of the original subexpressions can be derived from \( (c_1, c_2, \ldots, c_l) \). The calculated Gröbner basis of this Ideal based on a certain monomial ordering would provide the building blocks of the subexpressions (library components) [PD01, PD03]. However in our
approach and the exposed subexpressions result in Gröbner basis which is the our entire polynomial ring. If we denote the decomposition of the the set of polynomial(s) \( \{f_1, f_2, \ldots, f_n\} \) using polynomial division and a set of library components \( \{c_1, c_2, \ldots, c_l\} \) as \( \{f_1, f_2, \ldots, f_n\} \xrightarrow{\{c_1, c_2, \ldots, c_l\}} \) then the resulting polynomials after this step can be defined as:

\[
F = \{f_1, f_2, \ldots, f_n\} \\
F^{\{c_1, c_2, \ldots, c_l\}} = F \cup_{i=1}^{n} \cup_{j=1}^{q} \{c_j, \text{quo}(f_i, c_j), \text{rem}(f_i, c_j)\}
\]

### 2.2.5 Abstracted Algorithms

In this section we present abstracted algorithms that can be used to implement polynomial optimization formulated in the previous section. Our Optimization method deviates from these algorithms however the overall structure is closely followed. Algorithm 2.2.1 demonstrates the overall approach in polynomial simplification. We expose subexpressions first in the first line of the main loop. This is the most important step of the polynomial optimization and we will expand the theory and techniques in the following chapter and the overall outline of the algorithm is presented in algorithm 2.2.2. Next in line 3, we find compatibility of the exposed subexpressions and get valid decomposition sequence set for each input polynomials \( A_i = \{(c_1, c_2, \ldots), (c_1, c_2, \ldots), \ldots\} \) which is outlined in algorithm 2.2.3. Next in line 5, we select the best subexpressions based on these valid decomposition sequences and a given cost function (algorithm 2.2.4). The next step would be to decompose the input polynomials based on these extracted components. Algorithm 2.2.3 tries all the \( m \)-tuple of the indexes \( \{1, 2, \ldots, m\} \) [Knu11], which requires \( m! \) different scenario to consider. However this exponential growth in the optimization is avoided due to the fact that the exposed subexpressions of one input polynomial cannot always divide other input polynomials (not compatible). For each \( m \)-tuple, we iterate through all the indexes (1..m) and whenever
Algorithm 2.2.1 OPTIMIZE-POLYNOMIAL$(F)$

**Input:** $F_{inp} = \{f_1, f_2, \ldots, f_n\} \subset \mathbb{Z}[x_1, x_2, \ldots, x_k]$  
**Output:** $F = \{f_1, f_2, \ldots, f_v\} \subset \mathbb{Z}[x_1, x_2, \ldots, x_k]$  

1: $C \leftarrow$ SUBEXPRESSIONS$(F)$  
2: for all $f_i \in F$ do  
3: $A_i \leftarrow$ VALID-SEQUENCES$(f_i, C)$  
4: end for  
5: $A \leftarrow$ FIND-BEST$(F, \{A_1, A_2, \ldots, A_n\})$  
6: $F \leftarrow \emptyset$  
7: for all $f_i \in F_{inp}$ do  
8: $F_{cur} \leftarrow \{f_i\}$  
9: for $c_j \in A_i$ do  
10: for $f_{tmp} \in F_{cur}$ do  
11: $F_{cur} \leftarrow F_{cur} \cup \{c_j, \text{quo}(f_{tmp}, c_j), \text{rem}(f_{tmp}, c_j)\} \setminus \{f_{tmp}\}$  
12: end for  
13: end for  
14: $F \leftarrow F \cup F_{cur}$  
15: end for  
16: return $F$

Algorithm 2.2.2 SUBEXPRESSIONS$(F)$

**Input:** $\{f_1, f_2, \ldots, f_n\} \subset \mathbb{Z}[x_1, x_2, \ldots, x_k]$  
**Output:** $\{c_1, c_2, \ldots, c_m\} \subset \mathbb{Z}[x_1, x_2, \ldots, x_k]$  

: return $\{c_1, c_2, \ldots, c_m\}$

we cannot decompose the polynomial set (which includes all the polynomials generated in previous decomposition steps) with the component $c_i$ associated with that index we terminate that sequence and add that as a valid decomposition sequence to the return set $A$. The conditional statement at the line 8 makes sure that we do not return an empty ordered set which means we cannot decompose the $f$ with $c_{a_1}$. This component comes from another input polynomial and cannot be used as the first decomposition of $f$. In our approach this step of the formulation is part of the subexpression selection method since we rely on factorization and during the optimization step we expose this through matrix representation of possible factorizations. Algorithm 2.2.4 similarly goes through all the options
Algorithm 2.2.3 VALID-SEQUENCES($F$)

Input: $f \in \mathbb{Z}[x_1, x_2, \ldots, x_k]$
Input: $\{c_1, c_2, \ldots, c_m\} \subset \mathbb{Z}[x_1, x_2, \ldots, x_k]$
Output: $\{(a_1^{[1]}, a_2^{[1]}, \ldots), (a_1^{[2]}, a_2^{[2]}, \ldots), \ldots\}\,\{a_i^{[j]}\ldots \in [1, m]\$

1: $A \leftarrow \emptyset$
2: for all $(a_1^{[1]}, a_2^{[1]}, \ldots, a_m^{[m]}) \in m\text{-TUPLE}([1, 2, \ldots, m])$ do
3:     $F \leftarrow \{f\}$
4:     for all $i \leftarrow 1..m$ do
5:         if $\exists p \in F|\text{quo}(p, c_{a_i}) > 0$ then
6:             $F \leftarrow F \setminus \{p\} \cup \{\text{quo}(p, c_{a_i}), \text{rem}(p, c_{a_i}), c_{a_i}\}$
7:         else
8:             if $i > 1$ then
9:                 $A \leftarrow A \cup \{(a_1^{[1]}, a_2^{[1]}, \ldots, a_{i-1})\}$
10:            end if
11:        break
12:    end if
13:    end for
14: end for
15: return $A$

available for the $n$ input polynomials through the set of valid decomposition sequences $A_i$ for each polynomial. The number of options which the algorithm goes through is the product of number of valid decomposition sequences for each polynomial ($|A_1| \times |A_2| \times \cdots \times |A_n|$). For each of these options the algorithm decomposes each of the input polynomial based on the sequences and calculates a cost function ($\text{COST}$). It’s important to note that the $\text{COST}$ takes into account the shared components between the input polynomials. The minimum option is then chosen and returned as the best option. A naïve implementation of the function $\text{COST}$ is demonstrated in the algorithm 2.2.5. Algorithm 2.2.5 only considers the cost associated with multiplication and addition cost disregarding the common subexpressions; it does not take into the consideration whether the building blocks for the polynomial $f$ have been used before or not. The two constants $\text{COST}_{\text{ADD}}, \text{COST}_{\text{MULT}} \in \mathbb{R}$ are the cost of implementing addition and multiplication respectively. The algorithm $\text{COST}_{\text{POLY}}(f) \in \mathbb{R}$ returns the cost of the polynomial $f$ in terms of additions and multiplication and is shown in 2.2.6. The algorithm goes
Algorithm 2.2.4 FIND-BEST($F, A$)

Input: $F = \{f_1, f_2, \ldots, f_n\} \subset \mathbb{Z}[x_1, x_2, \ldots, x_k]$

Input: $\{A_1, A_2, \ldots, A_n\} | A_i = \{(a_1^i, a_2^i, \ldots), (a_1^2, a_2^2, \ldots)\}$

Output: $\{(b_1^1, b_2^1, \ldots), (b_1^2, b_2^2, \ldots), \ldots, (b_1^n, b_2^n, \ldots)\}$

1: $min\_cost \leftarrow \infty$
2: for all $(b_1^i, b_2^i, \ldots), \ldots, (b_1^n, b_2^n, \ldots) \in n\_TUPLE(\{A_1, A_2, \ldots, A_n\})$ do
3: $cost \leftarrow 0$
4: for all $i \leftarrow 1..n$ do
5: $cost \leftarrow cost + COST(f_i, (b_1^i, b_2^i, \ldots))$
6: end for
7: if $cost < min\_cost$ then
8: $best \leftarrow \{(b_1^1, b_2^1, \ldots), \ldots, (b_1^n, b_2^n, \ldots)\}$
9: end if
10: end for
11: return $best$

through the decomposition sequence and every time a portion of the polynomial is replaced by polynomial division of the form $c_{a_i} \times \text{quo}(p, c_{a_i}) + \text{rem}(p, c_{a_i})$ we subtract the cost of the replaced polynomial (line 5) and add the cost of the multiplication, factor and quotient as well as an addition and the remainder (if the remainder is greater than zero) in lines 6..12. The generated polynomials are then added to the set for the next iteration. This implementation decomposes the given polynomial $f$ based on the sequence and returns the cost of the implementation. In a more elaborate implementation which is used in section 3, the common subexpressions generated throughout these decompositions are only considered once. This approach has its drawbacks too since we don’t consider the fan-out of the circuits in the hardware implementation. However for the abstraction level used for the arithmetic optimization these simplifications are acceptable. Algorithm 2.2.6 demonstrates how we calculate the cost of implementing a polynomial. The number of additions directly comes from the number of terms subtracted by 1 (since addition is performed between the terms) (line 1). For each term we calculate the cost of multiplication by considering the degree of the term subtracted by 1 (line 4), as well as the coefficient of the term if it’s more than 1 (lines 5..7).
Algorithm 2.2.5 COST($f, A$)

**Input:** $f \in \mathbb{Z}[x_1, x_2, \ldots, x_k]$

**Input:** $A = (a_1, a_2, \ldots) | a_i \in [1..m]$ 

**Output:** $\text{cost} \in \mathbb{R}$ 

1: $\text{cost} \leftarrow \text{COST}_\text{POLY}(f)$
2: $F \leftarrow \{f\}$
3: for all $a_i \in (a_1, a_2, \ldots)$ do
4: select $p \in F$, $\text{quo}(p, c_{a_i}) > 0$
5: $\text{cost} \leftarrow \text{cost} - \text{COST}_\text{POLY}(p)$
6: $\text{cost} \leftarrow \text{cost} + \text{COST}_\text{POLY}(c_{a_i})$
7: $\text{cost} \leftarrow \text{cost} + \text{COST}_\text{MULT}$
8: $\text{cost} \leftarrow \text{cost} + \text{COST}_\text{POLY}(\text{rem}(p, c_{a_i}))$
9: if $\text{rem}(p, c_{a_i}) > 0$ then
10: $\text{cost} \leftarrow \text{cost} + \text{COST}_\text{ADD}$
11: $\text{cost} \leftarrow \text{cost} + \text{COST}_\text{POLY}(\text{rem}(p, c_{a_i}))$
12: end if
13: $F \leftarrow F \setminus \{p\} \cup \{\text{quo}(p, c_{a_i}), \text{rem}(p, c_{a_i}), c_{a_i}\}$
14: end for
15: return $\text{cost}$

The case of $-1$ as the coefficient is also not considered as a multiplication since the cost is considered in addition and no multiplication is performed.

### 2.3 Summary

Arithmetic datapath designs implement a series of \texttt{add}, \texttt{mult} operations over bit-vectors and they are generally modeled at RTL or behavioral-level as systems of multivariate polynomials of finite degree. In this chapter, we presented the previous work in polynomial optimization, with the objectives to reduce area cost, power, or delay. Unlike previous attempts, we formulated the problem using abstract algebra and presented the steps required to perform this task. Previous attempts can be mapped to these steps and our approach discussed in the following chapter rely on the same formulation. Finally we presented abstracted algorithms required to implement optimization approach that adheres to our formulation.
Algorithm 2.2.6 COST\textsubscript{POLY}(f)

Input: $f \in \mathbb{Z}[x_1, x_2, \ldots, x_k]$

Output: $cost \in \mathbb{R}$

1: $cost \leftarrow (|f| - 1) \times \text{COST}_{\text{ADD}}$
2: for all $t_i \in \text{terms}(f)$ do
3: if $\deg(t_i) > 0$ then
4: $cost \leftarrow cost + (\deg(t_i) - 1) \times \text{COST}_{\text{MULT}}$
5: if $|(coeff)(t_i)| > 1$ then
6: $cost \leftarrow cost + \text{COST}_{\text{MULT}}$
7: end if
8: end if
9: end for
10: return $cost$
CHAPTER 3

The Proposed Method

Our method is an extension of the method in [HFK06] as mentioned before. The main elements of this approach is to use factorization to find all possible factorizations, to find multiple-term (cube in [HFK06]) subexpressions, and finally to apply single-term subexpression elimination. Methods in [HKF05, HFK06] are ported from similar methods from boolean expression optimization [BM82, BRS87, RV92] which deal with thousands of literals and variables. This method considers coefficients and variables as literals hence it does not consider possibility of decomposition of the coefficients into their divisors. Although factorization of coefficients is proposed in [GK09] as a separate step in their optimization, we will show the shortcomings of their approach. Another issue with previous attempts has been the lack of class of the polynomials under study. In this research we present a class of polynomials used in modeling of non-linear systems, Volterra series (chapter 5.1). By focusing on this class we propose another optimization possible due to the fact that some variables are delayed inputs which allows us to save the products at the current clock cycle to be used in subsequent clocks. All these optimizations are done at the algebraic level. When it comes to arithmetic implementation we will provide optimizations as well (section 4). We avoid using the terminologies from in [HKF05, HFK06] since they are imported from the boolean expression optimization and we try to use terms suited for abstract algebra formulation. However, as a reference we provide the corresponding terms when we define ours. Our assumption is that we use integer coefficients and variables in our
polynomials and as mentioned before the domain of evaluating these polynomials is the polynomial ring defined over integer ring \( \mathbb{Z} \). Following are some definitions and we refer the reader to section A.1 for further reading on abstract algebra definitions.

**Definition 4.** \( \mathbb{Z}[x_1, \ldots, x_k] \) is the set of all polynomials defined with the set of variables \( \{x_1, \ldots, x_k\} \) over \( \mathbb{Z} \) (coefficients and variables in \( \mathbb{Z} \)).

**Definition 5.** Given polynomials in \( \mathbb{Z}[x_1, \ldots, x_k] \), a monomial is a product of the form \( x_1^{d_1} \cdots x_n^{d_n} \) where \( d_i \in \mathbb{Z}^+ \) (\( \mathbb{Z}^+ \) is the non-negative integers set).

**Definition 6.** Given polynomials \( \mathbb{Z}[x_1, \ldots, x_k] \), a term is a product of a coefficient \( c \) (in \( \mathbb{Z} \)) and a monomial. (Defined as cube in [HFK06])

### 3.1 Possible Factorizations

In this step we find all the possible ways to factor the set of input polynomials. To allow coefficient factorization we modify the definition of what is known as “cube-free” in Kernel/Co-Kernel method as following:

**Definition 7.** Polynomial \( f \in \mathbb{Z}[x_1, \ldots, x_k] \), is monomial-free if the greatest common divisor of all the terms is a constant \( c \in \mathbb{Z} \). In other words there is no variable \( x_i \) that can divide all the terms \( (\exists x_i | x_i | f) \).

This definition allows us to have these two factorization for \( f = 6x^2 + 4xy \):

\[
f = x(6x + 4y) \\
f = 2x(3x + 2y)
\]

Where both \( 6x + 4y \) and \( 3x + 2y \) are monomial-free whereas in [HFK06] only \( 6x + 4y \) is cube-free. We define multi-factor and factor extensions to terms kernel and co-kernel in [HFK06] as following:
Definition 8. Given a polynomial \( f \in \mathbb{Z}[x_1, \ldots, x_k] \) and a term \( t \), the quotient of the division of \( f \) by \( t \) (\( \text{quo}(f, t) \)) is called a multi-factor if it is monomial-free and has at least 2 terms. The term used to find a multi-factor is called a factor.

In other word, given a multi-factor \( c = \text{quo}(f, t) \), we cannot factor out any variable from \( c \). However in our definition (monomial-free) we can still factor out the coefficients. We represents the number of terms in polynomial \( f \) as \( |f| \).

The polynomial division used here is insensitive to the monomial ordering. This division is different from multivariate polynomial division and defined as following.

Definition 9. Given a polynomial \( f \in \mathbb{Z}[x_1, \ldots, x_k] \) and a term \( t \), we define \( q = \text{quo}(f, t) \), the quotient of the division of \( f \) by \( t \), and \( r = \text{rem}(f, t) \), the remainder of the division of \( f \) by \( t \), so that \( f = qt + r \) where \( t \) cannot divide any term in \( r \).

We refer to this division as division over term.

Definition 9 can be thought as a special case of multivariate polynomial division where the divisor is a term instead of a polynomial and the monomial ordering does not matter.

Definition 10. Given a polynomial \( f \in \mathbb{Z}[x_1, \ldots, x_k] \), we call the factorization of the form \( f = t \times q + r \) minimal if \( q \) is monomial-free.

Since our definitions have changed we present and modify the proofs for theorems in [HFK06] which are direct extension to account for coefficients divisors factorization.

Theorem 3.1.1. All minimal factorization can be obtained using the set of multi-factors and factors of an expression.

Proof. Given the polynomial \( f \in \mathbb{Z}[x_1, \ldots, x_k] \) and a minimal factorization \( f = t \times q + r \) (\( \exists x_i \mid x_i \mid q \)), we have to prove that we have a multi-factor which includes \( q \) with the corresponding factor \( t \). Let \( \{t_i\} \) be the set of original terms in \( f \) covering
Let \( \sum t_i = q \times t \). Let \( \{ t_j \} \) be the set of all other terms not overlapping \( \{ t_i \} \) which can also be divided by \( t \) \( (\forall t_j \mid t \mid t_j) \). If we define the polynomial \( g = \sum t_i + \sum t_j \), we have:

\[
g = t \times q + t \times \sum t'_j = t \times (q + \sum t'_j)
\]

Where \( t'_j = t_j/t \). We know the greatest common divisor of terms in \( q \) is a constant (monomial-free) so the greatest common divisor of terms in \( q + \sum t'_j \) is also a constant \( (\gcd(c_1, c_2x_1^{d_1} \ldots x_m^{d_n}) = c_3 \mid c_i \in \mathbb{Z}) \). Hence, \( q + \sum t'_j \) is monomial-free and a multi-factor of \( f \) with corresponding factor \( t \). Since \( q \) is part of this multi-factor we have proved the theorem. \( \square \)

This theorem shows that we can find all the minimal factorizations using the set of all multi-factors and factors.

**Theorem 3.1.2.** There exists multiple-term common subexpression among a set of polynomials if and only if there exists multiple-term intersection among the multi-factors of the polynomials.

**Proof.** The if direction is trivial. If \( q \) is multiple-term intersection of multi-factor, then we know we have multiples of \( q \) among the set of polynomials. For only if case, let’s assume we have \( g \), a multiple-term common subexpression among the set of polynomials. If \( g \) is monomial-free, each instance of common subexpression \( g \) is part of a multi-factor of the corresponding polynomial according to theorem 3.1.1. Hence, the intersection of the set of these multi-factors results in the common subexpression \( g \). If \( g \) is not monomial-free, then let’s have \( t \) the greatest common divisor of terms of \( g \) and \( g' = g/t \). The expression \( g' \) is monomial-free and according to the same argument above, we can find \( g' \) through intersection of some of the multi-factors. This proves the existence of intersection among the
multi-factors.

This theorem provides a method to find common subexpressions among a set of polynomials using intersection of the set of all multi-factors. As will be explained, this intersection results in a set of two or more common terms among some of the multi-factors. Algorithm 3.1.1 extracts the set of all multi-factors with their corresponding factors given a set of polynomials and a set of factoring elements \((E)\) which consists of the original input variables, added variables, coefficients and a subset of the coefficient divisors \((C)\). We do not need to include all divisors as will be explained. The returned set represents all possible factorization and is an extension to the similar algorithm in [HFK06] by considering the divisors of the coefficients. Algorithm 3.1.1 returns a set of tuples containing multi-factor and corresponding factor. The algorithm iterates through all the polynomials in the input set and in addition to the factorization possible, it adds each polynomial as a multi-factor with 1 as its factor. This is due to the fact that a polynomial or subset of its terms can be part of another polynomial without any factorization. The main part of the possible factorization happens in algorithm 3.1.2 (FACTORS). The algorithm 3.1.2 iterates through all the factorization elements (variables, coefficients and coefficients divisors) and we are only concerned with factorizations resulting in multi-factors (line 3). We use a greatest common divisor algorithm to find the common factor between the terms of the resulting multi-factor. Our \texttt{GCD} algorithm avoids factoring coefficients to avoid having coefficient factorization in

\begin{algorithm}
\caption{FIND-FACTORS\((F, E)\)}
\begin{algorithmic}
\STATE Input: \( F \subset \mathbb{Z}[x_1, x_2, \ldots, x_k, \ldots] \)
\STATE Input: \( E = \{x_1, x_2, \ldots, x_k, \ldots\} \cup \{c | c \in \mathbb{Z}\} \)
\STATE Output: \( \{(m_1, c_1), \ldots, (m_l, c_l)\} \subset \mathbb{Z}[x_1, x_2, \ldots, x_k, \ldots] \times \mathbb{Z}[x_1, x_2, \ldots, x_k, \ldots] \)
\STATE 1: \( M \leftarrow \emptyset \)
\STATE 2: \textbf{for all} \( f \in F \) \textbf{do}
\STATE 3: \hspace{1em} \( M \leftarrow M \cup \{(f, 1)\} \cup \text{FACTORS}(0, f, 1, E) \)
\STATE 4: \textbf{end for}
\STATE 5: \textbf{return} \( M \)
\end{algorithmic}
\end{algorithm}

26
Algorithm 3.1.2 FACTORS($i, f, factor_{imp}, E$)

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$M \leftarrow \emptyset$</td>
</tr>
<tr>
<td>2</td>
<td>for all $j \in [i,</td>
</tr>
<tr>
<td>3</td>
<td>if $\lfloor \text{quo}(f, E[j]) \rfloor \geq 2$ then</td>
</tr>
<tr>
<td>4</td>
<td>$factor_{multi} \leftarrow \text{quo}(f, E[j])$</td>
</tr>
<tr>
<td>5</td>
<td>$factor_{gcd} = \text{GCD}(factor_{multi})$</td>
</tr>
<tr>
<td>6</td>
<td>$\text{Seen} \leftarrow \text{false}$</td>
</tr>
<tr>
<td>7</td>
<td>for all $k \in [0, j]$ do</td>
</tr>
<tr>
<td>8</td>
<td>if $(E[k] \in \mathbb{Z} \land E[k] = \lfloor \text{coeff}(factor_{gcd}) \rfloor) \lor (E[k] \notin \mathbb{Z} \land E[k]\lfloor factor_{gcd} \rfloor)$ then</td>
</tr>
<tr>
<td>9</td>
<td>$\text{Seen} \leftarrow \text{true}$</td>
</tr>
<tr>
<td>10</td>
<td>break</td>
</tr>
<tr>
<td>11</td>
<td>end if</td>
</tr>
<tr>
<td>12</td>
<td>end for</td>
</tr>
<tr>
<td>13</td>
<td>if $\neg \text{Seen}$ then</td>
</tr>
<tr>
<td>14</td>
<td>$factor_{multi} \leftarrow \text{quo}(factor_{multi}, factor_{gcd})$</td>
</tr>
<tr>
<td>15</td>
<td>$factor \leftarrow factor_{imp} \times E[j] \times factor_{gcd}$</td>
</tr>
<tr>
<td>16</td>
<td>$M \leftarrow M \cup {(factor_{multi}, factor)} \cup \text{FACTORS}(j, factor_{multi}, factor, E)$</td>
</tr>
<tr>
<td>17</td>
<td>end if</td>
</tr>
<tr>
<td>18</td>
<td>end if</td>
</tr>
<tr>
<td>19</td>
<td>end for</td>
</tr>
<tr>
<td>20</td>
<td>return $M$</td>
</tr>
</tbody>
</table>

two places (since we perform in lines 3..4 in the main loop). The next step is to avoid factorizations seen before (lines 6..13) which has two different logic for coefficients and variables. For coefficients and their divisors are checked against the coefficient of the greatest common divisor of the terms. If the factorization is new we divide the multi-factor with the gcd and we add the gcd to the factor side. This means we have to multiply the input factor from the last call to the factorization, the element used in this iteration $E[j]$ and the gcd calculated (lines 14..15). The tuple of multi-factor and the factor is added to the set and we continue factorization by calling the factor recursively with the multi-factor as the input polynomial. The index $j$ is passed on to avoid previous factored
elements. The greatest common divisor algorithm (algorithm 3.1.3) as mentioned

Algorithm 3.1.3 GCD(F)

Input: $f \in \mathbb{Z}[x_1, x_2, \ldots, x_k, \ldots]$
Input: $E = \{x_1, x_2, \ldots, x_k, \ldots\} \cup \{c|c \in \mathbb{Z}\}$
Output: $gcd \in \mathbb{Z}[x_1, x_2, \ldots, x_k, \ldots]$

1: $\text{mon}_{gcd} \leftarrow 0$
2: $\text{coeff}_{gcd} \leftarrow 0$
3: $\text{coeff}_{min} \leftarrow \infty$
4: for all $t \in \text{terms}(f)$ do
5: $\text{mon}_{gcd} \leftarrow \text{gcd}(\text{mon}_{gcd}, \text{monomial}(t))$
6: $\text{coeff}_{gcd} \leftarrow \text{gcd}(\text{coeff}_{gcd}, \text{coeff}(t))$
7: if $|\text{coeff}(t)| < \text{coeff}_{min}$ then
8: $\text{coeff}_{min} \leftarrow |\text{coeff}(t)|$
9: end if
10: if $|\text{coeff}_{gcd}| < \text{coeff}_{min}$ then
11: $\text{coeff}_{gcd} \leftarrow 1$
12: end if
13: return $\text{coeff}_{gcd} \times \text{mon}_{gcd}$
14: end for
15: return $M$

before avoids divisor factorization with the following logic: If the gcd of all terms
has a coefficient smaller than the minimum coefficient of the terms we change
the coefficient to 1. This forces the main loop of FACTORS to perform the coeffi-
cient divisor factorization. So far we have shown that to generate all the possible
factors we need to pass on the set of polynomials and the factoring elements to
FIND-FACTORS. The set of factoring elements, in addition to the variables and
coefficients, includes coefficient divisors. However we prove in the following the-
orem that we need the greatest common divisor of corresponding terms and not
all divisors of the coefficients.

Theorem 3.1.3. There exists multiple-term common subexpression among a set
of polynomials if and only if there exists multiple-term intersection among the
multi-factors with the greatest common divisor of the coefficients factored out.

Proof. For the if case, let’s have $g$, the multiple-term intersection of multi-factors
with greatest common divisor of the coefficients factored out, then we know the
greatest common divisor of terms of $g$ is 1. This is a special case of monomial-
free definition and according to theorem 3.1.2 we know we have a multiple-term
common subexpression among polynomials corresponding to $g$. For the only if
case, lets assume we have $g$, a multiple-term common subexpression among the set
of polynomials. We define $c$ to be the greatest common divisor of the coefficients
of $g$ and $g' = g/c$. According to theorem 3.1.2 we can find $g'$ through intersection
of some multi-factors. We know the greatest common divisor of coefficients of $g'$
is 1, hence the greatest common divisor of coefficients of multi-factors have to be
1 (multi-factors are of the form $g' + \ldots$). This is only possible if original terms
$\{t_i\}$ corresponding to each instance of $g$ are factored as $\sum t_i = t \times (g' + \ldots)$ where
the coefficient of $t$ is the greatest common divisor of the coefficients of $\{t_i\}$. □

This theorem allows us to limit the divisors we would need to add to our fac-
toring elements and that is the greatest common divisor of coefficients of possible
common subexpressions or their corresponding multi-factors. However we need
to define the set of factoring elements before we can find the set of all multi-
factors/factors and following theorems provide the solution.

**Lemma 3.1.4.** Set of Non-negative Integers $\mathbb{Z}^+$ with the greatest common divisor
operation forms a monoid.

**Proof.** We lists the axioms needed for a monoid:

- **Closure** - This comes from the definition of gcd where $\gcd(a, b) \in \mathbb{Z}^+$

- **Associativity** - This comes from the properties of gcd operation where
  \[ \gcd(a, \gcd(b, c)) = \gcd(\gcd(a, b), c) \]

- **Identity Element** - Zero is the identity element under gcd operation since
  \[ \gcd(a, 0) = \gcd(0, a) = 0. \] We also define $\gcd(0, 0) = 0$ to complete this.
Even though this lemma is defined for the set of non-negative integers, any subset of it that is closed under gcd operation forms a monoid as well.

**Theorem 3.1.5.** Given monoid \((M, \text{gcd})\), where \(M\) is a superset of the set of coefficients of a polynomial \(f\), then \(M\) includes the greatest common divisor of coefficients of any subset of the polynomial \(f\).

*Proof.* Let’s assume polynomial \(f\) has the coefficients \(C = \{c_1, \ldots, c_i\}\). We have to prove that for any subset \(C_i\) of the terms \((C_i \subset C)\) we have \(\text{gcd}(C_i) \in M\). We prove this by induction. For the base case we have \(|C_j| = 2\) which is trivial, since the monoid is closed under the monoid operation and for the two terms we have \(\text{gcd}(c_1, c_2) \in M\) if \(c_1, c_2 \in C_j\). For the inductive step, we assume for \(|C_j| = n\) we have \(\text{gcd}(C_j) \in M\). For the case \(|C_k| = n + 1\), and any \(c_l \in C_k\) we have \(\text{gcd}(C_k \setminus \{c_l\}) \in M\). We also know \(\text{gcd}(C_k) = \text{gcd}(\text{gcd}(C_k \setminus \{c_l\}), c_l)\). We know the monoid is closed under the monoid operation, therefore \(\text{gcd}(C_k) \in M\). That completes the proof.

This theorem proves that our factoring elements set would need to include a set which is a superset of the polynomial coefficients and it should be closed under the binary greatest common divisor operation. We have presented the algorithm 3.1.4 to create the set of factoring elements. As mentioned before the coefficient factorization proposed in [GK09] has drawbacks since it’s a separate step from the subexpression extraction. Coefficient factorization is only considered for the case where a coefficient is multiples of another coefficient. For example for the polynomial \(8x + 12y\), they do not consider coefficient factorization, where in reality \(4(2x + 3y)\) can expose \(2x + 3y\) as a potential common subexpression used in other terms/polynomials. The other issue with their approach is that once the common coefficient is factored out, the resulting expression is directly considered
for decomposition. For example a factorization of the form $4(2x + 3y + \ldots)$ is only considered when $(2x + 3y + \ldots)$ can be used. Our approach integrates coefficient factorization into the generation of multi-factors/factors which allows the optimizing step to consider different factorization based on coefficients and their divisors and in the above case also consider all subsets of $(2x + 3y + \ldots)$ with at least two terms. The algorithm 3.1.4, for each polynomial adds the coefficients of all the terms to the set (line 3) and then loops until we have found the gcd of all possible combination (lines 4..11). Algorithm only terminates when the set does not change anymore (line 11). This makes sure that our set is closed under gcd operation. We keep this operations separate $(M)$ for each polynomial, since the factorization is performed on individual polynomials. Finally we need to include the polynomial ring variables as the factorization elements (line 14) and remove 1 from the factoring elements (line 15).

```
Algorithm 3.1.4 ELEMENTS($F$)
Input: $F \subset \mathbb{Z}[x_1, x_2, \ldots, x_k, \ldots]$
Output: $E = \{x_1, x_2, \ldots, x_k\} \cup \{c|c \in \mathbb{Z}\}$
1: $E \leftarrow \emptyset$
2: for all $f \in F$ do
3: $M \leftarrow \text{coeffs}(f)$
4: repeat
5: $M' \leftarrow M$
6: for all $i \in [0, |M| - 1]$ do
7: for all $j \in [i + 1, |M|[] do
8: $M \leftarrow M \cup \{\gcd(M[i], M[j])\}$
9: end for
10: end for
11: until $M' = M$
12: $E \leftarrow E \cup M$
13: end for
14: $E \leftarrow E \cup \{x_1, x_2, \ldots, x_k\}$
15: $E \leftarrow E \setminus \{1\}$
16: return $E$
```
3.2 Subexpressions Compatibility Test

We perform two different optimization: multiple-term and single-term common subexpression elimination. This step is integrated into the optimization step in the actual implementation as will be explained in the next section. However we present this to conform with our presented formulation. Our multiple-term optimization focuses on the multi-factors generated in the subexpression extraction step. We would need to intersect different multi-factors in the optimization step to reduce the overall cost of our system. As will be explained in optimization step we would use minimum weighed rectangular covering problem which are inherently hard to solve. For each multi-factor generated we would need to consider the combination of 2 or more of its terms to calculate the intersection with other multi-factors as a common subexpression. Given two subexpressions $c_i$ and $c_j$ we assume here that they both come from factorizations of the same polynomial since in each iteration we want to factor out common subexpressions in each polynomial which result in the highest reduction of the overall cost. The factor and multi-factor that these common subexpressions are coming from are represented as $\text{multi-factor}(c)$ and $\text{factor}(c)$. In actual implementation these are pointers to the tuple corresponding to that factorization. The compatibility check becomes:

$$\text{compatible}(f, c_i, c_j) = \begin{cases} 
False, & \text{if } |\text{terms}(c_i \times \text{factor}(c_i)) \cap \text{terms}(c_j \times \text{factor}(c_j))| > 0 \\
True, & \text{otherwise}
\end{cases}$$

In other words, the two common subexpressions cannot have terms generated from the same term of the original polynomial. For example if following is the
polynomial in a set of input polynomials with its two factorizations:

\[ f_i = 2x_1x_3 + 3x_2x_3 + 2x_1x_4 \ldots \]
\[ = x_3(2x_1 + 3x_2 + \ldots) + \ldots \]
\[ = 2x_1(x_3 + x_4 + \ldots) + \ldots \]

Then the two subexpressions \(2x_1 + 3x_2\) and \(x_3 + x_4\) are not compatible since they have \(2x_1x_3\) term in common form the original polynomial. So if one is selected the other cannot be used in the same valid decomposition sequence. Similarly for single-term optimization we would need to consider these compatibilities. However unlike the multiple-term, the common subexpression (in this case common term) only affects a single term in each polynomial. So in each iteration we can only modify a term once. Once the polynomials are written with new defined variables (decomposition step) we would revisit the affected terms again for a possible common term elimination.

### 3.3 Delayed Terms

Previous attempts to polynomial optimization have not focused on any specific class of polynomials. They have only relied on several extracted benchmarks with limited applicability. Our research has focused on a class of polynomials, Volterra polynomials (series), which is used for modeling nonlinear systems. In chapter 5 we expand on Volterra polynomials (section 5.1) and their applications. For the purpose of this section it’s suffice to mention that the polynomial variables consist of some variables and their delayed values. We represent the delayed variables as \(x_{i[d]}\) where \(i\) is the index of the input variable and \(d\) represents the number of clock delays. For the current value of an input \((d = 0)\) we simply refer to the input as \(x_i\). For example a system with two inputs and 2 memory locations for each
input, has 6 variables and defined over $\mathbb{Z}[x_1, x_{1[1]}, x_{1[2]}, x_2, x_{2[1]}, x_{2[2]}]$. With this representation we can see that some terms can be calculated in a cycle and be used in the next cycle. We use $z^{-1}$ as the delay element similar to the common practice in digital signal processing theory [PM96]. For example, the system mentioned above have following delayed terms equivalencies:

$$
Z^{-1}x_1^{d_1}x_2^{d_2} = x_{1[1]}^{d_1}x_{2[1]}^{d_2}
$$

$$
Z^{-1}x_{1[1]}^{d_1}x_{2[1]}^{d_2} = x_{1[2]}^{d_1}x_{2[2]}^{d_2}
$$

$$
Z^{-1}x_1^{d_1}x_{2[1]}^{d_2}x_2^{d_3} = x_{1[1]}^{d_1}x_{2[1]}^{d_2}x_{2[2]}^{d_3}
$$

where $d_i \in \mathbb{Z}^+$. For example $x_1^3x_2^2$ if stored for one cycle will provide the value for $x_{1[1]}^3x_{2[2]}^2$. This allows us to expose more common subexpressions by considering these equivalencies. As will be explained in the optimization step, we only consider equivalencies of single common terms. This can be equally applied to multiple-term common subexpressions.

### 3.4 Polynomials Rewriting

Since we use ring of polynomials defined over integers ($\mathbb{Z}[x_1, \ldots, x_k]$), we need to clarify the rewriting step where we introduce new variables ($u_1, \ldots$) and the effect of it in the original polynomial ring. All of our algorithms work on polynomials even though we have a set of non-linear equations. Hence, the left-hand side variable is not part of the operations involved and they are merely a name defined for the polynomial on the right-hand side. When we find a common subexpression and assign a new variable to it, we first replace all instances of the expression in the original set of equations with the new variable. Additionally we add a new equation to the set where we associate the common subexpression to the new variable. The polynomial ring that we use at that point has an additional
variable. For example if we have a set of polynomials defined over $\mathbb{Z}[x_1, \ldots, x_k]$ by adding a new variable $u_1$ and rewriting the polynomials, we will have a set of polynomials defined over the variables $\{x_1, \ldots, x_k, u_1\}$. We know $\mathbb{Z}[x_1, \ldots, x_k] \subset \mathbb{Z}[x_1, \ldots, x_k, u_1]$ so the original ring is a subring of the new ring. Hence, in our formulation based on polynomial rings, addition of a new variable is equivalent to ring extension.

3.5 Polynomial Optimization

As explained before our approach has two steps: multiple-term and single-term common subexpression elimination. According to the formulation from section 2.2 we would have all the possible valid decomposition sequences. However we avoid the exponential complexity of this by focusing at each iteration to find the best option for decomposition (for either multiple/single term optimization). Instead once the decomposition steps are performed, we go back through all these steps with the new variables added until there is no common subexpression left. This greedy approach avoids the double exponential time required for checking every possible decomposition order. As will be explained, by randomizing this selection and running the algorithm multiple times, we find better decompositions. We use formulation used in [HKF05, HFK06] for this step with modifications required for delayed products and coefficients divisors factorizations. Each section explains these two optimizations which include the 4 steps from our formulation in each iteration.

3.5.1 Multiple-Term Optimization

This step is similar to the algorithm from [HKF05] with the exception that we will consider all possible factorization including the divisors of the coefficients. In each iteration we find the possible factorizations as a set of ordered pairs
\{(m_1, c_1), \ldots\} (section 3.1) for each polynomial in the input set. The goal is to find a common subexpression with multiple terms where by factoring it out, we have the maximum decrease in the overall cost. As explained in our subexpression compatibility test (section 3.2) we cannot choose two common subexpressions with terms resulted from factorization of the same terms in the original polynomials.

We formulate the optimization step with a matrix called Multiple-Term Matrix (MTM) similar to the Kernel Cube Matrix (KCM) in [HFK06]. Different naming is to distinguish our method since we have used different terminology with our enhanced method and we have included common coefficient divisors factorizations as well. MTM is formed so that the rows are associated with each ordered pair multi-factor/factor and the columns are the distinct terms generated in all multi-factors: We have a value 1 in for an element of this matrix if the associated column is a term in the associated row’s multi-factor. The goal is to find a set of rows and columns with the elements associated with each row and column of these sets with value 1 that will result in the highest reduction in cost. We represent the set of indices of the rows selected as \(R\) and the set of indices of the columns selected as \(C\). The set of all possible factorizations (ordered pairs) is \(M\), so \(R\) consists of indices in \([1, |R|]\). Similarly if we have an ordered set of the distinct terms in all multi-factors as \(D\) we have:

\[
\text{REDUCTION}(M, D, R, C) = \sum_{i=1}^{|R|} \text{COST}_\text{POLY} \left( \sum_{j=1}^{|C|} (\text{factor}(M[R[i]]) \times D[C[j]]) \right) - \left( (|C| - 1)\text{COST}_\text{ADD} + \sum_{j=1}^{|C|} \text{COST}_\text{POLY}(D[C[j]]) \right) - \left( |R|\text{COST}_\text{MULT} + \sum_{i=1}^{|R|} \text{COST}_\text{POLY}(\text{factor}(M[R[i]])) \right) \tag{3.1}
\]

As mentioned before \(\text{COST}_\text{POLY}\) calculates the cost of a polynomial by counting the additions and multiplications weighted by their cost. This reduction is the cost of
the original terms (first row) subtracted by the cost of the terms from the common subexpression (columns) accounted once and then multiplied by each row’s factor as well as the cost of that factor. Algorithm 3.5.1 shows the steps required to find in each iteration the best multiple-term subexpression and a new polynomial in the input set is generated for this subexpression. Then the set of polynomials is rewritten with the new variable associated with the subexpression. Hence we have polynomials defined as \( F \subseteq \mathbb{Z}[x_1, x_2, \ldots, x_k, \ldots] \). This algorithm in each iteration find all possible factorizations (line 3) and generates the distinct list of terms in multi-factors (lines 5–8) as well as the MTM matrix (lines 9). `valid-selection` selects all the valid selections in MTM in which all the elements associated with rows and columns selected have value of 1 (line 12). To avoid exponential runtime, at each iteration we select the best common subexpression which may not result in the best overall decomposition. We have randomized the order in which we iterate valid-selections and by running the algorithm multiple times we arrive at better solutions. On average we ran this algorithm 2 to 3 times to arrive at our results. The reduction resulted by each selection is calculated based on equation 3.1 (lines 12–19). Using the rows and columns resulting in maximum reduction, we create a new variable which is defined as the sum of all distinct terms associated with these columns (lines 22–24). Finally the new variables definition \( f_{new} \) is added to the input set of polynomials \( F \). The new variable \( x_{new} \) is added to the list of factorization elements \( E \). The polynomials associated with each row of the maximum selection are updated with the new variable (lines 28–30). Finally the MTM is updated for the next iteration. This step enforces the compatibility of the multiple-term subexpression we select in each inner loop iteration. This is done by updating all the elements associated with the terms updated in the original polynomials to 0. Inner loop continues until we can’t find any valid selection in MTM. The outer loop continues every time we add a new variable (extract a subexpression).
Algorithm 3.5.1 MULTIPLE-TERM\((F, E)\)

Input: \( F \subseteq \mathbb{Z}[x_1, x_2, \ldots, x_k, \ldots] \)

Input: \( E = \{x_1,x_2,\ldots,x_k,\ldots\} \cup \{c | c \in \mathbb{Z}\} \)

Output: \( F \subseteq \mathbb{Z}[x_1, x_2, \ldots, x_k, \ldots] \)

1: repeat
2: \( \text{changed} \leftarrow \text{False} \)
3: \( M \leftarrow \text{FIND-FACTORS}(F, E) \)
4: if \( |M| > 0 \) then
5: \( D \leftarrow \emptyset \)
6: for all \( m \in M \) do
7: \( D \leftarrow D \cup \text{factor}(m) \)
8: end for
9: \( MTM \leftarrow \text{MTM}(M, D) \)
10: repeat
11: \( \text{reduction}_{\text{max}} \leftarrow 0 \)
12: for all \( R, C \in \text{valid-selection}(MTM) \) do
13: \( \text{reduction} \leftarrow \text{REDUCTION}(M, D, R, C) \)
14: if \( \text{reduction} > \text{reduction}_{\text{max}} \) then
15: \( \text{reduction}_{\text{max}} \leftarrow \text{reduction} \)
16: \( C_{\text{max}} \leftarrow C \)
17: \( R_{\text{max}} \leftarrow R \)
18: end if
19: end for
20: if \( \text{reduction}_{\text{max}} > 0 \) then
21: \( \text{changed} \leftarrow \text{True} \)
22: \( f_{\text{new}} \leftarrow 0 \)
23: for all \( j \in C_{\text{max}} \) do
24: \( f_{\text{new}} \leftarrow f_{\text{new}} + D[C[j]] \)
25: end for
26: \( F \leftarrow F \cup \{f_{\text{new}}\} \)
27: \( E \leftarrow E \cup \{x_{\text{new}}\} \)
28: for all \( i \in R_{\text{max}} \) do
29: \( F[R[i]] \leftarrow F[R[i]] \times \text{factor}(M[R[i]]) \times x_{\text{new}} \)
30: end for
31: \( MTM \leftarrow \text{UPDATE-MTM}(M, D) \)
32: end if
33: until \( \text{reduction}_{\text{max}} = 0 \)
34: end if
35: until \( \text{changed} = \text{False} \)
3.5.2 Single-Term Optimization

The goal of this step is to find single term common subexpressions similar to compilers’ subexpression elimination [jua] and cube intersection step in [HFK06]. Our method considers the delayed terms as explained in section 3.3. We extend the formulation in [HFK06] where a matrix is formed to find the best single term subexpression. The rows of this matrix has all the terms in the set of polynomials (not distinct) and the columns are associated with the factoring elements. The value of the matrix element is the power of the column’s factoring element in the term associated with that row. We call this matrix the Single-Term Matrix (STM). The goal similar to Multiple-Term Optimization is to find a valid set of rows and columns where the elements are greater than 0 where by selecting it we have the maximum reduction in cost. We represent the set of indices of the rows selected as \( R \) and the set of indices of the columns selected as \( C \). If the set of all terms is \( T \), \( R \) consists of indices in \([1, |T|]\). Similarly \( C \) consists of indices in \([1, |E|]\) (where \( E \) is the list of all factoring elements). The common term would be the greatest common divisor of all the terms (rows) in the selection. This means we would need to find the minimum value of each column for the selected rows and that is the power of that factoring element in the gcd. Hence the reduction becomes:

\[
\text{REDUCTION}(T, E, R, C) = \sum_{i=1}^{\left|R\right|} \text{COST}_\text{POLY}(R[i]) - \left( \text{COST}_\text{POLY}(\gcd(R)) \right)
- \left( \sum_{i=1}^{\left|R\right|} \text{COST}_\text{POLY}(\frac{R[i]}{\gcd(R)}) + \text{COST}_\text{MULT}R \right)
\]  

(3.2)

This reduction is the cost of the original terms (first row) subtracted by the cost the gcd and the multiplications still needed for the remaining powers of factoring elements for each term (row). Algorithm 3.5.2 shows the steps required to find
in each iteration the best single-term subexpression. A new polynomial is defined as the gcd for the terms and a new variable is defined to be associated with this definition. The terms are rewritten with this new variable. This algorithm in each iteration generates the list of all terms (lines 4.12) and all the delayed possibilities of that term as defined in section 3.3. Then it generates the STM matrix (line 13). valid-selection selects all the valid selections in STM in which all the elements associated with rows and columns selected have values (power) greater than 0 (line 14). To avoid exponential run-time, at each iteration we select the best common subexpression which may not result in the best overall decomposition. The cost of this selection is exponential in degree of the terms, which at this point after the multiple-term optimization is at most 2. We have randomized the order in which we iterate valid-selections and by running the algorithm multiple times we arrive at better solutions. On average we ran this algorithm 2 to 3 times to arrive at our results. The reduction resulted by each selection is calculated based on equation 3.2 (lines 14..20). Using the rows and columns resulting in maximum reduction, we create a new variable which is defined as the gcd of all terms involved (all the rows) (lines 22..25). Finally the new variables definition \( f_{new} \) is added to the input set of polynomials \( F \). The new variable \( x_{new} \) is added to the list of factorization elements \( E \). The polynomials associated with each row of the maximum selection are updated with the new variable (lines 27..36). In the case where the common subexpression is from a delayed term (line 28), the original term (gcd) is found and added to the list of polynomials (lines 29..30). We also need to define a new variable for this original un-delayed gcd (line 30). We do not remove the delayed variable \((x_{new})\) since other rows may rely on it. Finally we need to keep track of these delayed terms in set \((S)\) of ordered pairs associating the original variable \( x_{new}^* \) to its delayed variable \( x_{new} \). This association will be used later in the implementation section to generate the delay (memory) elements for these products. The main loop continues until we can’t find any
Algorithm 3.5.2 SINGLE-TERM($F, E$)

Input: $F \subset \mathbb{Z}[x_1, x_2, \ldots, x_k, \ldots]$
Input: $E = \{x_1, x_2, \ldots, x_k, \ldots\} \cup \{c | c \in \mathbb{Z}\}$
Output: $F \subset \mathbb{Z}[x_1, x_2, \ldots, x_k, \ldots]$
Output: $S = \{(x_i, x_j), \ldots\}$

1: $S \leftarrow \emptyset$
2: repeat
3: $\textit{reduction}_{\text{max}} \leftarrow 0$
4: $T \leftarrow ()$
5: for all $f \in F$ do
6: for all $t \in \text{terms}(f)$ do
7: append($T, t)$
8: for all $t_d \in \text{delayed}(t)$ do
9: append($T, t_d$)
10: end for
11: end for
12: $\text{STM} \leftarrow \text{STM}(T, E)$
13: for all $R, C \in \text{valid-selection}(\text{STM})$ do
14: $\textit{reduction} \leftarrow \text{REDUCTION}(T, E, R, C)$
15: if $\textit{reduction} > \textit{reduction}_{\text{max}}$ then
16: $\textit{reduction}_{\text{max}} \leftarrow \textit{reduction}$
17: $C_{\text{max}} \leftarrow C$, $R_{\text{max}} \leftarrow R$
18: end if
19: end for
20: if $\textit{reduction}_{\text{max}} > 0$ then
21: $f_{\text{new}} \leftarrow 0$
22: for all $i \in R_{\text{max}}$ do
23: $f_{\text{new}} \leftarrow \gcd(f_{\text{new}}, R[i])$
24: end for
25: $F \leftarrow F \cup \{f_{\text{new}}\}$, $E \leftarrow E \cup \{x_{\text{new}}\}$
26: for all $i \in R_{\text{max}}$ do
27: if is-delayed($R[i]$) then
28: $f_{\text{new}} \leftarrow \text{reverse-delay}(f_{\text{new}})$
29: $F \leftarrow F \cup \{f_{\text{new}}^*\}$, $E \leftarrow E \cup \{x_{\text{new}}^*\}$
30: $F[R[i]] \leftarrow F[R[i]] - R[i] + \frac{R[i]}{f_{\text{new}}} \times x_{\text{new}}$
31: $S \leftarrow S \cup \{(x_{\text{new}}^*, x_{\text{new}})\}$
32: else
33: $F[R[i]] \leftarrow F[R[i]] - R[i] + \frac{R[i]}{f_{\text{new}}} \times x_{\text{new}}$
34: end if
35: end for
36: end if
37: until $\textit{reduction}_{\text{max}} = 0$
38: return $F, S$
valid selection in STM. For example, we demonstrate the transformations on a set of polynomials \{y_I, y_Q\} from small set of terms of the polynomials from the application presented in section 5 defined over \(\mathbb{Z}[x_I, x_{I[1]}, x_Q, x_{Q[1]}]\). The subscript \(I\) and \(Q\) represent the in-phase/quadrature (real/imaginary) parts of the inputs and outputs of the system:

\[
\begin{align*}
y_I &= 3x_I x^2_{I[1]} + 2x_Q x^2_{I[1]} - 3x^3_{I[1]} - 3x^2_{I[1]} x_{Q[1]} \\
y_Q &= -4x_I x^2_{I[1]} + x_Q x^2_{I[1]} + 3x^3_{I[1]} - 3x^2_{I[1]} x_{Q[1]}
\end{align*}
\]

These two polynomials require 23 multiplications. After finding the factors and applying factorization to reduce the cost of implementation we will have:

\[
\begin{align*}
u_1 &= x_I - x_{I[1]} - x_{Q[1]} \\
u_2 &= -4x_I + x_Q + 3u_4 \\
u_3 &= 2x_Q + 3u_1 \\
u_4 &= x_{I[1]} - x_{Q[1]} \\
y_I &= x^2_{I[1]} u_3 \\
y_Q &= x^2_{I[1]} u_2
\end{align*}
\]
After the common subexpression elimination with delayed products the overall polynomials system becomes:

\[
\begin{align*}
    u_1 &= x_l - x_{l[1]} - x_{Q[1]} \\
    u_2 &= -4x_l + x_Q + 3u_4 \\
    u_3 &= 2x_Q + 3u_1 \\
    u_4 &= x_{l[1]} - x_{Q[1]} \\
    u_5 &= x_{l[1]}^2 \\
    y_l &= u_3u_5 \\
    y_Q &= u_2u_5
\end{align*}
\]

Our method has resulted in reduction of multiplications required from 23 in the original set \(\{y_l, y_Q\}\) to 7 in the polynomial set \(\{u_1, u_2, u_3, u_4, u_5, y_l, y_Q\}\). As will be explained in the implementation chapter (section 4), the multiplication by multiple of 2 is implemented with shifts and the multiplication by 3 is implemented with shift and addition. Therefore, we will have 3 multiplications in the actual implementation. Another example demonstrating the benefit of considering common coefficients is the set of polynomials \(\{f, g\}\):

\[
\begin{align*}
    f &= 9x^2y^2 + 6xy^3 + 6x \\
    g &= 6x^2y^2 + 4xy^3 + 2y
\end{align*}
\]
defined over $\mathbb{Z}[x, y, z]$ with 18 multiplications. [HFK06] approach would result in the following decomposition:

\[
\begin{align*}
    u_1 &= 9x + 6y \\
    u_2 &= 6x + 4y \\
    u_3 &= y^2u_1 + 6 \\
    u_4 &= xyu_2 + 2 \\
    f &= xu_3 \\
    g &= yu_4
\end{align*}
\]

resulting in 10 multiplications. Our approach results in the following decomposition:

\[
\begin{align*}
    u_1 &= u_4 + u_6 \\
    u_2 &= yu_5 + 2 \\
    u_3 &= xu_5 + 1 \\
    u_4 &= 3x \\
    u_5 &= yu_1 \\
    u_6 &= 2y \\
    f &= u_2u_4 \\
    g &= u_3u_6
\end{align*}
\]

Our approach even though in this case does not consider delayed products results in 7 multiplications. As mentioned before and will be explained in chapter 4 the multiplication by 2 and 3 are not implemented by multipliers. Therefore we require 5 multiplications to implement this polynomial set. Our approach performs better due to the fact that we consider coefficient divisors factorization as well. Another
example of extraction and usage of delayed products is the set of polynomials \( \{ f, g \} \subset \mathbb{Z}[x_1, x_2, x_{1[1]}, x_{2[1]}] \):

\[
\begin{align*}
f &= 9x_1^2x_{2[1]}^2 + x_{1[1]} \\
g &= x_{1[1]}^3
\end{align*}
\]

There is no multiple-term common subexpression to expose and after applying single-term optimization we have:

\[
\begin{align*}
u_1 &= x_1^2 \\
u_2 &= \# u_1 \\
f &= 9x_{2[1]}^2u_1 + x_{1[1]} \\
g &= x_{1[1]}u_2
\end{align*}
\]

where \( u_2 \) is the delayed value of the product saved in \( u_1 \) and \( \# \) represents a memory element, as will be further explained. The set of polynomials generated are represented as a Data-Flow Graph such as the one displayed in Figure 3.1 for the polynomial set \( \{ i = u_1 + u_2 + u_3; u_1 = a^2; u_2 = 5a1; u_3 = \# u_1; \} \). The DFG of the polynomials allow us to map it to different architectures such as Software, Data-Flow Processors, Application-Specific Integrated Circuit (ASIC), or FPGA. We focus on implementation of these polynomials on FPGA’s. Using the DFG we can find the sizes needed for each variable generated (\( u_1, u_2, \ldots \)) as well as the system output. In the implementation section we further enhance these ideas by using expression trees.

### 3.6 Results

We applied our method to the set of polynomials resulted by the expansion of the Volterra series of a non-linear system. We present this application in chapter
Figure 3.1: Sample Data-Flow Graph

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Multiplication (%)</th>
<th>Addition (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSE</td>
<td>78</td>
<td>62</td>
</tr>
<tr>
<td>Horner</td>
<td>66 (−15.4%)</td>
<td>62 (0.0%)</td>
</tr>
<tr>
<td>[HFK06]</td>
<td>51 (−34.6%)</td>
<td>53 (−14.5%)</td>
</tr>
<tr>
<td>Our Approach</td>
<td>46 (−41.0%)</td>
<td>54 (−12.9%)</td>
</tr>
</tbody>
</table>

Table 3.1: Number of Arithmetic Operations

5. The set of polynomials \( \{ y_I, y_Q \} \) defined over \( \mathbb{Z}[x_I, x_{I[1]}, x_Q, x_{Q[1]}] \) is listed in equations 5.5 and 5.6 in section 5. We also applied the polynomial optimization techniques such as Horner method, common-subexpression elimination (CSE), and [HKF05, HFK06] (Kernel/Co-Kernel + CSE). Table 3.1 demonstrates the number of operations involved in different approaches. We see reductions of 41.0% and 12.9% in multiplication and addition operations comparing to common subexpression elimination outperforming other methods. We also see better decrease of resources usage of FPGA implementation comparing to other methods. We used
the data-flow graphs of each approach to generate the bit-widths required for each operation. We synthesized these approaches for Altera Cyclone V 5CEFA7F31C7 FPGA using Quartus 13 and simulated using Multisim. Table 3.2 lists out the number of DSP’s and adaptive logic modules (ALM) required for each implementation. Our method will perform at least as good as the approach in [HFK06] since we consider all the common subexpressions in their methods as well as additional possible subexpressions which include subexpressions resulted from coefficient factorization and delayed products. The randomization of the selection at each iteration has resulted in the most optimized solution in 2 to 3 runs on average for this application. To test our approach further, we generated additional 4 sets of polynomials by changing the parameters of our model. This was an attempt to validate the advantage of our approach over previous attempts to more than one set of inputs. As will be explained, the predistorter (the studied application) is affected by temperature and in an actual implementation, the designer might have to include the predistorter for different temperature ranges. Table 3.3 lists the multiplication involved in the different set of polynomials generated by modifying the parameters. Section 5.2 lists the parameter values used. The first row reflects the polynomials associated with the original parameters used and listed in Table 3.1 above. As seen our approach outperforms previous attempts in decreasing the number of multiplications.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>DSP</th>
<th>ALM</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSE</td>
<td>35</td>
<td>1015</td>
</tr>
<tr>
<td>Horner</td>
<td>54</td>
<td>637</td>
</tr>
<tr>
<td>[HFK06]</td>
<td>24</td>
<td>377</td>
</tr>
<tr>
<td>Our Approach</td>
<td>22</td>
<td>274</td>
</tr>
</tbody>
</table>

Table 3.2: FPGA Implementation Costs
<table>
<thead>
<tr>
<th>Poly. Inputs</th>
<th>CSE</th>
<th>Horner (%)</th>
<th>[HFK06] (%)</th>
<th>Our Approach (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Param. Set 1</td>
<td>78</td>
<td>66(−15.4%)</td>
<td>51(−34.6%)</td>
<td>46(−41.0%)</td>
</tr>
<tr>
<td>Param. Set 2</td>
<td>78</td>
<td>66(−15.4%)</td>
<td>51(−34.6%)</td>
<td>47(−39.7%)</td>
</tr>
<tr>
<td>Param. Set 3</td>
<td>76</td>
<td>64(−15.8%)</td>
<td>50(−34.2%)</td>
<td>43(−43.4%)</td>
</tr>
<tr>
<td>Param. Set 4</td>
<td>74</td>
<td>64(−13.5%)</td>
<td>46(−37.8%)</td>
<td>39(−47.3%)</td>
</tr>
<tr>
<td>Param. Set 5</td>
<td>80</td>
<td>68(−15.0%)</td>
<td>55(−31.3%)</td>
<td>51(−36.3%)</td>
</tr>
</tbody>
</table>

Table 3.3: Number of Multiplications of Different Impl. of Predistorter

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Application</th>
<th>$t/n/d$</th>
<th>Original</th>
<th>Horner</th>
<th>[HFK06]</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mibench</td>
<td>Automotive</td>
<td>8/3/2</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Savitzky-Golay</td>
<td>Image Process.</td>
<td>9/2/3</td>
<td>20</td>
<td>11</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>DIRU</td>
<td>Image Process.</td>
<td>8/2/4</td>
<td>18</td>
<td>12</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>PSK</td>
<td>Digital Comm.</td>
<td>9/2/4</td>
<td>25</td>
<td>16</td>
<td>12</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 3.4: Benchmarks

3.6.1 Common Benchmarks

We applied our method to a set of benchmarks used previously. The variables in these benchmarks are independent and are not delayed values of each other. Hence, we could not apply the delayed product technique. Table 3.4 shows the number of multiplication required for each benchmark. The first and second column specify the benchmark’s name and the application it’s used in. The third column shows the number of terms, number of variables, and the degree of the polynomial. The benchmarks include an automotive benchmark from [GRE01], Savitzky-Golay (SG) filter from image processing and used in [GK07, SAF09], Digital Image Rejection Unit (DIRU) used in [AF09], Phase-Shift Keying (PSK) from digital communication and used in [AF09, PD03]. We have compared our results to Horner and the method in [HFK06]. Our method works as good or better than the approach in [HFK06]. As mentioned earlier, for these benchmarks, we only rely on our coefficient divisor factorization in addition to the techniques in [HFK06]. Since these benchmarks are not Volterra polynomials, we could not use the delayed product technique of the section 3.3. The benchmarks that we do not see any improvement over Kernel/Co-Kernel method, is due to lack of greatest
common divisors between coefficients or no cost reduction if common coefficient is factored out.

3.7 Summary

In this chapter we presented the high-level algebraic methods we have implemented to optimize a set of polynomials. At each step we expose common subexpressions and we select the best option which lowers the implementation cost. Multiple-Term optimization method generates all the possible factorization including coefficient factorization. Set of terms that would result in the highest cost reduction are selected as the common subexpression. The Single-Term optimization method finds a common term among all terms at each step. We have focused on the Volterra series as our polynomial class. This has allowed us to expose delayed products, which are possible due to the fact that some variables are delayed values of other variables. These optimizations have resulted in reduction of 41.0% and 12.9% in number of multiplication and addition in comparison with common subexpression elimination technique. We also generated 4 sets of polynomials by changing our model parameters to ensure the advantage of using our methods. We also applied our method to a set of benchmarks from different domains such as automotive, image processing, and communication systems. We saw decrease in number of multiplications even though delayed product technique is not applicable to these benchmarks.
CHAPTER 4

Implementation

In previous chapters we have presented the high-level algebraic manipulation of polynomials to achieve lower cost of implementation. In this chapter we attempt to use digital arithmetic techniques known with contributions from us to optimize the hardware evaluation of these polynomials further. The results reported in the previous chapter relied on the synthesis tool Quartus [Qua] to provide the implementations required for the high-level expressions. Synthesis tools are mostly geared towards general logic optimization and they have limited capability built-in towards optimization of mathematical expressions. For example, the operations are mapped to library components which use conventional arithmetic lacking possibility of more advanced and faster operations using redundant number systems [EL03]. We have taken the optimization further by trying to generate the operations involved in the polynomial evaluation using redundant number arithmetic. To avoid the hassle and challenges of manually designing and assembling different components needed, we present a generator that we have implemented to generate the Hardware Description Language (HDL) of the polynomials under test. We use Verilog [Pal03] as our HDL of choice, However, the generator can be easily modified to generate other languages.

4.1 Redundant Digit Arithmetic

In conventional arithmetic the operands and the output are represented in conventional form, using conventional digit set (e.g. radix-2 \{0, 1\}). This simple
approach has the drawback of requiring a carry propagation adders (CPA) which in best case has $O(\log n)$ time complexity [EL03]. Redundant number representation allows faster operations and result in constant time complexity regardless of the operands’ precision. The drawback of using redundant form is the increase of the circuit size and a non-conventional representation of operands and results. In our polynomial evaluation, the output needs to be represented in the conventional form, so we can keep the internal variables and computations in redundant form to speed up the evaluation. There are two major redundant forms: Sign-Digit (SD) and Carry-Save (CS) representations. In this research we focus and use CS form. The CS form of a $n$-bit number $v$ is a set of two $n$-bit numbers, $vc$ and $vs$, where the sum of them equals $v$ (Figure 4.1). In our case we use signed integers so the redundant representation consists of two 2’s complement numbers. Since $c$ and $s$ can have different values with the sum still equal to $v$, this representation is redundant. As explained before, the drawback of this representation is the increase in size where in this case is twice. Using this representation carry-save adders avoid propagating the carry. A carry-save adder, shown in Figure 4.1, can be used to perform the addition of three numbers in constant time ($\tau_{FA}$). By keeping the result in this redundant form we have avoided the delay dependent on the precision. By combining carry-save adders we can perform addition of any number of operands. We will present a $[4:2]$ adder block which we will be using in addition to this $[3:2]$ adder block.

<table>
<thead>
<tr>
<th>$n-1$</th>
<th>$n-2$</th>
<th>…</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$vc$</td>
<td>●</td>
<td>○</td>
<td>…</td>
<td>○</td>
</tr>
<tr>
<td>$vs$</td>
<td>●</td>
<td>○</td>
<td>…</td>
<td>○</td>
</tr>
<tr>
<td>$v$</td>
<td>●</td>
<td>○</td>
<td>…</td>
<td>○</td>
</tr>
</tbody>
</table>

Table 4.1: Carry-Save (CS) Representation
4.1.1 Multi-Operand Addition

Adding several operands, in the case of conventional arithmetic, if broken down to a adder-tree, would require logarithmic delay for the tree and logarithmic delay for each addition. To avoid addition operation delay, we have used CS representation. We can combine FA’s to create multi-operand additions. A lot of research has gone over creating building blocks [EL96, Wal64, Wei81, Lim78]. Figure 4.2 demonstrates a [4 : 2] adder design where we have used two layers of [3 : 2] adders (FA). This design is simple however has $2\tau_{FA}$ delays. A better design is demonstrated in...
Figure 4.3 where the delay is $1.5\tau_{FA}$ [EL03] if implemented as application-specific integrated circuit (ASIC). We have used this design throughout our implementation. In this section we only consider the case where the operands arrive at the same time (for example in the case of multiplication and partial products). When we explain our generator we will provide an algorithm where we consider the arrival time of the inputs. By combining $[3 : 2]$ and $[4 : 2]$ adders we can find the combinations resulting in lowest delay for different number of operands. Figure 4.2 demonstrates the adder-trees generated for 5 to 16 inputs, where □ and ■ represent $[3 : 2]$ and $[4 : 2]$ adders respectively. What we have explained in this section is based on row-reduction techniques. There is also a technique which reduces the bit-matrix by columns [EL03].

### 4.1.2 Multiplications

We present what has been the standard practice for multiplication of two operands $p = x \times y = \sum_{i=0}^{n-1} 2^i x y_i$, where the result is represented in CS form and each of
the operands can be in 2’s complement or CS form and $y_i$ represents the $i^{th}$ bit of the multiplier. If we expand the sum, once we have partial products for all $i$’s $(2^i x y_i)$ we can use multi-operand addition as explained in the previous section to calculate the result in CS form. Since we only deal with signed integers in this research, we will explain the consideration of the sign-bit and unsigned arithmetic would be a simplified version of what we present. Since the multiplier $y$ is in 2’s complement form, the sign bit has a negative weight $y = -2^{n-1}y_{n-1} + \sum_{i=0}^{n-2}2^i y_i$. This means that the last row ($-2^{n-1}y_{n-1}x$) would need to be subtracted from the rest instead of added. Negation in 2’s complement form is calculate by logical negation and addition of 1 to it.

$$-2^{n-1}y_{n-1}x = 2^{n-1}y_{n-1}(-x)$$
$$= 2^{n-1}y_{n-1}(-x + 1)$$
$$= 2^{n-1}y_{n-1}x + 2^{n-1}y_{n-1}$$

Figure 4.4 shows the multiplication of two signed 4-bit integers. ● represents bits with negative weight. As seen the partial products are sign extended. To avoid the sign extensions which result in bigger circuits, we can remove the negative

<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>FA</td>
<td>2.5FA</td>
<td>2.5FA</td>
<td>3FA</td>
<td>3FA</td>
<td>3.5FA</td>
<td>4FA</td>
</tr>
<tr>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>FA</td>
<td>4FA</td>
<td>4FA</td>
<td>4.5FA</td>
<td>4.5FA</td>
<td>4.5FA</td>
<td>4.5FA</td>
</tr>
</tbody>
</table>

Table 4.2: Adder-Trees
Figure 4.4: Multiplication $4 \times 4$ Signed Integers

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$x$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$y$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$2^4xy_0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$2^3xy_1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$2^2xy_2$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$2^1xy_3$</td>
<td></td>
</tr>
<tr>
<td>$\neg s$</td>
<td>$-1$</td>
<td>$0$</td>
<td>$0$</td>
<td>$0$</td>
<td>$0$</td>
<td>$0$</td>
<td>$p_c$</td>
</tr>
<tr>
<td>$p_s$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

weight of the sign bit position [AR78]:

$$-s = -s + 1 - 1$$
$$= (1 - s) - 1$$
$$= \neg s - 1$$

Figure 4.5 shows this modification. We can combine the constant row ($-120$) and the row with $2^3y_3$ according to:

$$2^3(-x_0y_3 + y_3 + 1) = 2^4y_3 + 2^2\neg(x_0y_3)$$

In case where the adder blocks have different delays for different inputs are considered in [PZ91, PZ93]. Using the delay-matrices representing the propagation delay between the inputs and outputs, they have analytically presented the asymptotically optimal solution for the minimum delay multipliers adder trees. They have considered general adder blocks of the form $[2^n - 1 : n]$. Although their solution can be applied to any adder block, our adder block has similar delays for the outputs, hence their solution doesn’t offer improvements. In this case we can
use a [4 : 2] row reduction to produce the result in CS form.

**Radix-4 Multiplier**

To reduce the number of rows we can consider higher radix for the multiplier. We examine radix-4 representation where the rows become \( \times 1, \times 2, \) and \( \times 3 \) multiples of the multiplicand. To avoid the costly \( \times 3 \) multiple, we can recode the multiplier into minimally redundant radix-4 representation (\( \{ \overline{2}, \overline{1}, 0, 1, 2 \} \)) [EL96, EL03]. The recoding converts the \( y = \sum_{i=0}^{n-1} 2^i y_i \) into \( y = \sum_{i=0}^{\lfloor n/2 \rfloor - 1} 4^i z_i \), where \( z_i \in \{ \overline{2}, \overline{1}, 0, 1, 2 \} \). This recoder relies on overlapping 3-bit blocks from the multiplier bit vector. We present this recoder [EL03] without any modifications. Figure 4.6 demonstrates
the design of the recoding block for the $i^{th}$ digit ($z_i$) represented with $\text{sign}$, $\text{one}$, and $\text{two}$ signals. The delay of this block is $0.5\tau_{FA}$ and can be calculated in parallel for all bits. Since the multiplier is radix-4, each row is shifted two bit places to the left. Similar simplification applied to 2’s complement multiplier ($y$) for the sign-bits by changing the negative weight and addition of $\overline{1}$ is applied here [EL03].

**Redundant Multiplier**

If the multiplier is in the CS form we have:

$$x \times (y_c + y_s) = x \times y_c + x \times y_s$$

This would result in twice the number of rows of partial products. To avoid this, double recoding from CS to radix-4 [EL96, BL96] has been proposed. We present that and extend it to the case where CS form are signed (2’s complement). We need to modify the recoder for the sign-bit position. The goal of this recoder is to go from a digit set $v_i \in \{0, 1, 2, 3, 4, 5, 6\}$ (2 bits from each Carry and Sum vectors) to minimally redundant radix-4 digit set $p_i \in \{\overline{2}, \overline{1}, 0, 1, 2\}$. Figure 4.7 demonstrates the two steps involved where we go first from $v_i \in \{0, 1, 2, 3, 4, 5, 6\}$ to $q_i \in \{0, 1, 2, 3, 4, 5\}$ and then to $p_i \in \{\overline{2}, \overline{1}, 0, 1, 2\}$ where $q_i = z_i + h_i$ and

---

**Figure 4.6: 2’s complement to minimally redundant Radix-4 Recoder**

---

---
\[ p_i = w_i + t_i. \] Tables 4.3.a and 4.3.b list the values of these recoding variables. We present the bit-level implementation from [EL96]. We know \( v_i \) is the sum of 4 bits from CS representation (figure 4.8). We have:

\[
v_i = 2s_{2i+1} + s_{2i} + 2c_{2i+1} + c_{2i}
\]
Figure 4.8: CS bits partitioning

where $s$ and $c$ are the CS bit vectors. To satisfy the recoding for figure 4.3.a we have:

\[
h_{i+1} = \begin{cases} 
1, & \text{if } v_i \geq 4 \\
0, & \text{otherwise}
\end{cases}
\]

\[
z_i = v_i - 4h_{i+1}
\]

Similarly for figure 4.3.b we have:

\[
t_{i+1} = \begin{cases} 
1, & \text{if } q_i \geq 2 \\
0, & \text{otherwise}
\end{cases}
\]

\[
w_i = q_i - 4t_{i+1}
\]

We follow [EL03] and represent $z_i = 2z_1 + z_0$ with $z_1 \in \{0, 1\}$ and $z_0 \in \{0, 1, 2\}$. We represent $h_i$ with a bit, $z_1$ and $z_0$ as vectors $(z_1^1, z_1^0)$ and $(z_0^2, z_0^1, z_0^0)$ where $z_l^q$
is set when $z_l = q$. Recoding in figure 4.3.a becomes:

\[
\begin{align*}
    h_{i+1} &= s_{2i+1}c_{2i+1} \\
    z_1^1 &= s_{2i+1} \oplus c_{2i+1} \\
    z_0^0 &= \neg z_1^1 \\
    z_0^2 &= s_{2i}c_{2i} \\
    z_1^2 &= s_{2i} \oplus c_{2i} \\
    z_0^1 &= \neg (s_{2i} + c_{2i})
\end{align*}
\]

We represent $t_i$ with a bit and $w_i$ is represented by $(w_i^\overline{0}, w_i^\overline{1}, w_i^0, w_i^1)$ where again $w_i^q$ is set when $w_i = q$. The recoding in figure 4.3.b becomes:

\[
\begin{align*}
    t_{i+1} &= z_0^1 h_i + z_0^2 + z_1^1 \\
    w_i^\overline{0} &= z_0^2 z_1^0 h_i + z_0^1 z_1^0 h_i + z_0^0 z_1^1 \neg h_i \\
    w_i^\overline{1} &= z_0^2 z_1^0 h_i + z_0^1 z_1^0 h_i + z_0^0 z_1^1 h_i \\
    w_i^0 &= z_0^2 z_1^0 h_i + z_0^1 z_1^0 h_i + z_0^0 z_1^1 \neg h_i \\
    w_i^1 &= z_0^2 z_1^0 h_i + z_0^1 z_1^0 \neg h_i + z_0^0 z_1^1 h_i
\end{align*}
\]

Similarly $p_i$ is represented with $(p_i^\overline{0}, p_i^\overline{1}, p_i^0, p_i^1)$ where $p_i^q$ is set when $p_i = q$. Since $p_i^q$'s are mutually exclusive when none of the bits are set (0,0,0,0) we consider the case as $p_i = 0$. Finally the final conversion from $(t_i, w_i)$ to $p_i$ becomes:

\[
\begin{align*}
    p_i^\overline{0} &= w_i^\overline{0} \neg t_i \\
    p_i^\overline{1} &= w_i^\overline{0} t_i + w_i^\overline{1} \neg t_i \\
    p_i^0 &= w_i^1 \neg t_i + w_i^0 t_i \\
    p_i^1 &= w_i^1 t_i
\end{align*}
\]

Figure 4.9 demonstrates the implementation of this recoder block. Since in our
CS representation each vector is in 2’s complement form we now discuss the case of the sign-bit position. The goal of the recoder at this bit position \( m = \left\lceil \frac{n}{2} \right\rceil - 1 \) is to go from a digit set \( v_m \in \{4, 3, 2, 1, 0, 1, 2\} \) \( (c_{2m+1} \text{ and } s_{2m+1} \text{ have negative weight}) \) to minimally redundant radix-4 digit set \( p_m \in \{2, 1, 0, 1\} \). Similar to before we will go from \( v_m \in \{4, 3, 2, 1, 0, 1, 2\} \) to \( q_m \in \{0, 1, 2, 3, 4, 5\} \) and then to \( p_m \in \{2, 1, 0, 1\} \) where \( q_m = z_m + h_m \) and \( p_m = w_m + t_m \). Tables 4.4.a and 4.4.b list the values of these recoding variables. To satisfy the recoding for figure 4.4.a

<table>
<thead>
<tr>
<th>( v_m )</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h_{m+1} )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( z_m )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

(a) \( v_m \) to \( q_m = z_m + h_m \) recoding

<table>
<thead>
<tr>
<th>( q_m )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_{m+1} )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( w_m )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

(b) \( q_m \) to \( p_m = w_m + t_m \) recoding

Table 4.4: Sign Position Double Recoding
we have:

\[ h_{m+1} = \begin{cases} 1, & \text{if } v_m \leq 1 \\ 0, & \text{otherwise} \end{cases} \]

\[ z_m = v_m - 4h_{m+1} \]

Similarly for figure 4.4.b we have:

\[ t_{m+1} = \begin{cases} 1, & \text{if } q_m \geq 2 \\ 0, & \text{otherwise} \end{cases} \]

\[ w_m = q_m - 4t_{m+1} \]

We represent \( z_m = 2z_1 + z_0 \) with \( z_1 \in \{0, 1\} \) and \( z_0 \in \{0, 1, 2\} \). We represent \( h_{m+1} \) with a bit and it’s set if \( h_{m+1} = 1 \). We represent \( z_1 \) each \( z_0 \) as vectors \((z_1^1, z_0^1)\) and \((z_0^2, z_1^0, z_0^0)\) where \( z_q^q \) is set when \( z_q = q \). Recoding in figure 4.4.a becomes:

\[ h_{m+1} = s_{2m+1} + c_{2m+1} \]

\[ z_1^1 = s_{2m+1} \oplus c_{2m+1} \]

\[ z_0^0 = \neg z_1^1 \]

\[ z_0^2 = s_{2m}c_{2m} \]

\[ z_1^0 = s_{2m} \oplus c_{2m} \]

\[ z_0^0 = \neg (s_{2m} + c_{2m}) \]
The second recoding stays similar as before because we are going from \( q_m \in \{0, 1, 2, 3, 4, 5\} \) and then to \( p_m \in \{0, 1, 2, 3\} \):

\[
\begin{align*}
t_{m+1} &= z^1_0h_m + z^2_0 + z^1_1 \\
w^\pi_m &= z^2_0z^0_1h_m + z^1_0z^1_0h_m + z^0_0z^1_1h_m \\
w^\tilde{T}_m &= z^2_0z^0_1h_m + z^1_0z^1_1h_m + z^0_0z^1_1h_m \\
w^0_m &= z^2_0z^1_1h_m + z^1_0z^1_1h_m + z^0_0z^0_1h_m \\
w^1_m &= z^2_0z^1_1h_m + z^1_0z^0_1h_m + z^0_0z^0_1h_m \\
p^\pi_m &= w^\pi_m - t_m \\
p^\tilde{T}_m &= w^\pi_m t_m + w^\tilde{T}_m - t_m \\
p^1_m &= w^1_m - t_m + w^0_m t_m \\
p^2_m &= w^1_m t_m
\end{align*}
\]

Figure 4.10 demonstrates the implementation of this sign-bit recoder block. Since
the sign-bit digit has resulted in \( h_{m+1} \in \{1, 0\} \) and \( t_{m+1} \in \{0, 1\} \), we would need to calculate extra radix-4 digit \( (p_{m+1}) \). We know \( p_{m+1} = h_{m+1} + t_{m+1} \in \{1, 0, 1\} \) so \( p^\pi_{m+1} = 0 \) and \( p^2_{m+1} = 0 \) and we have:

\[
\begin{align*}
  p^\pi_{m+1} &= 0 \\
  p^T_{m+1} &= h_{m+1} - t_{m+1} \\
  p^1_{m+1} &= -h_{m+1} t_{m+1} \\
  p^2_{m+1} &= 0
\end{align*}
\]

To make the CS recoder similar to 2’s complement recoder we present the sign, one, and two signals as:

\[
\begin{align*}
  \text{sign} &= p^\pi_i + p^T_i \\
  \text{one} &= p^T_i + p^1_i \\
  \text{two} &= p^\pi_i + p^2_i
\end{align*}
\]

In the case of \( i \in [0, m] \) we can simplify \( \text{sign} \):

\[
\begin{align*}
  \text{sign} &= w^\pi_i - t_i + w^T_i t_i + w^T_i - t_i \\
  &= w^\pi_i + w^T_i - t_i
\end{align*}
\]

In the case of \( p_{m+1} \) we have:

\[
\begin{align*}
  \text{sign} &= p^T_{m+1} \\
  \text{one} &= p^T_{m+1} + p^1_{m+1} \\
  \text{two} &= 0
\end{align*}
\]

The recoders presented change the representation of the multiplier from CS or 2’s complement into minimally redundant radix-4 (\( \{2, 1, 0, 1, 2\} \)). The multiples are
generated as shown in Figure 4.11. The thick lines indicate bit vectors. The one and two signals select the positive multiple of the multiplicand. The sign bit, if set, inverts the bits of the resulting multiple. To complete the 2’s complement negation we need to also add the sign bit to the least significant bit (LSB) position of this multiple and pass it to the multi-operand addition that calculates the result of the multiplication. In case of a positive multiple (sign = 0), the positive multiple is added.

4.1.3 Constant Multiplication

Multiplication by a constant of the form \( p = x \times c = \sum_{i \in \{j|c_j \neq 0\}} 2^i x \), where the sum is iterated over the bit positions which have 1. This is due to the fact that when the constant bit is 0, the row becomes all zeros and it does not contribute to the sum. To reduce the number of rows, we can recode the constant into minimally redundant radix-4 representation as we presented before. We recode into a canonical form [EL03] where the constant is represented with the digit set \{\tilde{1}, 0, 1\}. This changes the \( n \)-bit constant operand with \( n/2 \) nonzero bits on average to a representation with \( n/3 \) nonzero digits on average [EL03]. This conversion only happens during the design or in our case during the code generation.
of our generator. Algorithm 4.1.1 represents the steps required. The input is the

Algorithm 4.1.1 CANONICAL(a)

Input: \( A = (a_{n-1}, \ldots, a_0) | a_i \in \{0, 1\} \)
Output: \( C = (c_{n-1}, \ldots, c_0) | c_i \in \{\overline{1}, 0, 1\} \)

1: \( C \leftarrow A \)
2: \( c_{n-1} \leftarrow -c_{n-1} \)
3: \( \text{cur}_\text{pos} \leftarrow -1 \)
4: \( \text{cur}_\text{digit} \leftarrow 0 \)
5: \textbf{for all} \( i \in [0, n-1] \) \textbf{do}
6: \textbf{if} \( c_i \neq \text{cur}_\text{digit} \) \textbf{then}
7: \textbf{if} \( i - \text{cur}_\text{pos} > 1 \land \text{cur}_\text{digit} \neq 0 \) \textbf{then}
8: \( \text{cur}_\text{pos} \leftarrow -\text{cur}_\text{pos} \)
9: \textbf{for all} \( j \in [\text{cur}_\text{pos}, i] \) \textbf{do}
10: \( c_j \leftarrow 0 \)
11: \textbf{end for}
12: \( c_i \leftarrow c_i + \text{cur}_\text{digit} \)
13: \textbf{end if}
14: \( \text{cur}_\text{pos} \leftarrow i \)
15: \( \text{cur}_\text{digit} \leftarrow c_i \)
16: \textbf{end if}
17: \textbf{end for}
18: \textbf{return} \( C \)

binary representation in an ordered set (list) \( A \). We make a copy of \( A \) into \( C \) where we change to a canonical representation (line 1). Since the input is in 2’s complement, the sign bit has negative weight so we change the bit sign of \( c_{n-1} \) (line 2). Two variables \( \text{cur}_\text{pos} \) and \( \text{cur}_\text{digit} \) are for tracking repetition of digits. Once we have tracked more than one of digit 1 in the input (line 7) we replace the consecutive 1’s with a 1 in the current position and \( \overline{1} \) at the first occurring (lines 8..11) and the rest of the bits are set to 0. To account for the sign bit position, the 1 is added to the current bit position instead of setting it to 1. This is for the case where we have for example \( \overline{1}1 \ldots 1 \) with \( 0 \ldots \overline{1} \) (line 12). Lines 14..15 initialize the consecutive 1’s.
4.1.4 Truncation

As will be explained later, when we generate the polynomial evaluator, we use interval arithmetic [BO97] to calculate the size of the variables and the internal nodes in our expression trees. This is to optimize the design by only calculating the results up to the required bit-vector size. For example, two variables $a$ and $b$ represented as 4-bit 2’s complement numbers are in the range $[-8, 7]$. According to interval arithmetic we have:

$$[\min_1, \max_1] \times [\min_2, \max_2] =$$
$$[\min\{\min_1 \times \min_2, \min_1 \times \max_2, \max_1 \times \min_2, \max_1 \times \max_2\},$$
$$\max\{\min_1 \times \min_2, \min_1 \times \max_2, \max_1 \times \min_2, \max_1 \times \max_2\}]$$

So for the product of $a$ and $b$ we have:

$$[-8, 7] \times [-8, 7] = [-56, 64]$$

A variable with the range $[-56, 64]$ requires 8 bits in 2’s complement representation. However $-a \times b$ is in range $[-64, 56]$ and only requires 7 bits. This allows us to truncate our variables and calculations. Another example is when we use the recoders to go to radix-4 we have to extend 1 bit position for odd number of bits in multiplier. If the multiplier is in CS form then the recoder also generates an extra row, so overall we would have extra 2 bits of output which are not necessary in the rest of evaluation. The truncation needs to consider sign extension of the CS form in following computations, for example if we have two 4-bit 2’s complement $c = -5 = (1011)_2$ and $s = 1 = (0001)_2$. The sum is $v = s + c = -4 = (1100)_2$ and requires 3 bits. However if we truncate most significant bits from $c$ and $s$ we have $c = 3 = (011)_2$ and $s = 1 = (001)_2$. Even though the sum still has the correct value ($v = s + c = (100)_2 = -4$), if we extend them the sum will not be correct.
For example by sign extending one bit, we have $c = (0011)_2$ and $s = (0001)_2$ and the sum will be $v = (0100)_2 = 4$. [Noi91] presents a correction to $[3 : 2]$ adders and truncation without a proof. We extend that idea since we use $[4 : 2]$ adders in the last part of array trees and provide a proof. Following theorem proves what needs to be done to truncate in the case of 1 extra bit of CS representation.

**Theorem 4.1.1.** Given a number $v = (v_n, \ldots, v_0)$ represented in Carry-Save representation as $s = (s_n, \ldots, s_0)$ and $c = (c_n, \ldots, c_0)$. If $v$ can be represented with $n$ bits then we can discard $s_n$ and $c_n$ with the following modifications:

\[
\begin{align*}
  c_{n-1} &\leftarrow c_{n-1} \oplus s_n \oplus c_n \\
  s_{n-1} &\leftarrow s_{n-1} \oplus s_n \oplus c_n
\end{align*}
\]

**Proof.** We know that $v$ can be represented as $(v_{n-1}, \ldots, v_0)$ (as an $n$ bit number) and it’s in the range $[-2^{n-1}, 2^{n-1} - 1]$. We also know that the sum of first $n - 1$ bits of $s$ and $c$ has the range of $(s_{n-2} : s_0) + (c_{n-2} : c_0) = [0, 2^{n-1}]$. Now we consider different cases of $s_n, s_{n-1}/c_n, c_{n-1}$:

- **10** - This case is not possible since $v = c + s = -2^n - 2^n + [0, 2^{n-1}] = [-4 \times 2^{n-1}, -3 \times 2^{n-1}] \cap [-2^{n-1}, 2^{n-1} - 1] = \emptyset$
- **11** - $v = c + s = -2^n - 2^n + 2^{n-1} + 2^{n-1} + [0, 2^{n-1}] = [-2^n, -2^{n-1}] \cap [-2^{n-1}, 2^{n-1} - 1] = -2^{n-1}$. In this case the truncation is fine, since we know we to have $v = -2^{n-1}$ we will have a carry into bit position $n - 1$. If later on we sign extend $c$ and $s$ this carry will be propagated in their sum, hence the calculation is correct.
- **11/10** - This case is not possible since $v = c + s = -2^n - 2^n + 2^{n-1} + [0, 2^{n-1}] = [-3 \times 2^{n-1}, -2 \times 2^{n-1}] \cap [-2^{n-1}, 2^{n-1} - 1] = \emptyset$
- **01** - This case is not possible since $v = c + s = 2^{n-1} + 2^{n-1} + [0, 2^{n-1}] = [2 \times 2^{n-1}, 3 \times 2^{n-1}] \cap [-2^{n-1}, 2^{n-1} - 1] = \emptyset$
- \( 00_{00} - v = c + s = [0, 2^{n-1}] \) and since \( v \) is non-negative, the truncation is fine. In later operation if we sign extend, we will be extending \( s_{n-1} = 0 \) and \( c_{n-1} = 0 \), keeping the sum a non-negative number.

- \( 01_{00}/00_{01} - 01_{00}/01_{00} - \) This case is not possible since \( v = c + s = 2^{n-1} + [0, 2^{n-1}] = [2^{n-1}, 2 \times 2^{n-1}] \cap [-2^{n-1}, 2^{n-1} - 1] = \emptyset \)

- \( 10_{00}/00_{10} - v = c + s = -2^n + [0, 2^{n-1}] = [-2^n, -2^{n-1}] \cap [-2^{n-1}, 2^{n-1} - 1] = -2^{n-1} \). In this case we know we have a carry onto bit position \( n-1 \). If we truncate, and later sign extend we will have a positive number. To avoid this if we write \( -2^n = -2^{n-1} - 2^{n-1} \), we can flip the bits at bit position \( n-1 \). This will allow the sum to propagate the carry coming into \( n-1 \) position.

- \( 10_{11}/01_{10} - v = c+s = -2^n+2^{n-1}+[0, 2^{n-1}] = [-2^{n-1}, 0] \cap [-2^{n-1}, 2^{n-1} - 1] = [-2^{n-1}, 0] \). This means there is a carry coming into bit position \( n-1 \) in which case by sign extending after truncation, we still propagate that carry. To simplify the final design, if we flip the bits in position \( n-1 \), we would not change the sum since only one of the bits is 1 and the other one is 0.

- \( 01_{11}/10_{01} - v = c+s = -2^n+2^{n-1}+2^{n-1}+[0, 2^{n-1}] = [0, 2^{n-1}] \cap [-2^{n-1}, 2^{n-1} - 1] = [0, 2^{n-1} - 1] \). The result is non-negative so there is no carry coming into position \( n-1 \) otherwise the sum would be negative. If we just truncate and then sign extend, we will have a negative number since we are extending \( s_{n-1} = 1 \) and \( c_{n-1} = 1 \). Again in this case we we flip the bits in \( n-1 \) position, we would avoid the sum to be negative in the case of a sign extension later on.

We showed that in cases where \( c_n \neq s_n \) we would need to flip the bits in position \( n-1 \) and then truncate so that later sign extension of \( s \) and \( c \) would result in the correct sign for the sum. Hence, if \( c_n \oplus s_n = 1 \) then we flip bits which requires
another XOR or:

\[ c_{n-1} \leftarrow c_{n-1} \oplus s_n \oplus c_n \]
\[ s_{n-1} \leftarrow s_{n-1} \oplus s_n \oplus c_n \]

Theorem 4.1.1 shows that we need to calculate only an extra bit in our adder trees and to truncate to the number of required bits we can use the transformation in this theorem. Going back to design of \([4 : 2]\) adder in figure 4.3 we see that due to the design, we require one additional bit. So for our result to be correct for the \(n\) bit result:

- Calculate up to \(n + 2\) bits resulting in Carry-Save vectors
- Update \(n - 1\) bit position by flipping these bits if bits in \(n\) position are different
- Truncate both bit-vectors (CS) to \(n\) bits

### 4.2 Generator

To simplify the implementation we have implemented a generator that automatically generates the Verilog code of the circuit that evaluates the set of input polynomials. The generator goes through series of phases:

- **Parser** - This step using a context-free grammar parses the input set of polynomials presented as a string of expressions and generates expression trees representing each statement in the grammar.

- **Expression Tree Modification** - This phase modifies the expressions trees
to correct and optimize evaluation by removing extra plus signs in multi-operand addition and generating multiplication trees.

- **Size Calculator** - This phase visits the expression trees to find the sizes of all variables in our polynomials. Depending on the implementation, we have to consider the representation of our variables (conventional or redundant in our case).

- **Code Generator** - This phase is the main phase where we start generating the output, Verilog code, representing the input polynomials.

We will present these phases in the subsequent sections. The Code Generator phase can be modified to map the polynomials to a different architecture or HDL language.

### 4.2.1 Parser

This is the first phase of our generator where we get the set of polynomials as input and we have to parse and generate expression trees, similar to compilers and interpreters [App98]. The table 4.5 shows the context-free grammar [Hop07] of the expected input. The overall structure of the input has set of statements of the form `<id>` `=' `<expr>` `;' (separated by semicolons). Left-hand side of each statement is a variable and the right-hand side has an expression (polynomial) defining that variable. The definitions are straight-forward except for the # where is generated in the polynomial optimization section. It represents a delay element which generates the required code in Verilog to save the value of `<id>` for the next cycle. This part of grammar was added to address delayed products defined in section 3.3. The definition of `<term>` also includes the negation of the term. This is defined for the case where the first term of the expression is negated. The output of this phase is a set of expression trees which will be used in the following phases. For example the graphical representation of the output of this phase for the set of
polynomials \{i = u_1 - u_2 + u_3; u_1 = a_2^2; u_2 = 5a_1a_2b_1; u_3 = \#u_1;\} is demonstrated in figure 4.12. There are some issues with these expression trees. First the sign of multi-operand addition in the case of positive (addition) is redundant. Multi-operand multiplication also needs to be addressed. As explained in the next section we use a expression tree visitor that will modify the expression trees to address these issues.
4.2.2 Expression Tree Modification

There are two issues as explained in the last section. In the figure 4.12 in the definition of \(i\) we see that \(u_3\) has a positive node before its addition to \(u_1\) and \(u_2\). Unlike \(u\) with a negation block this node is unnecessary. We also have the issue with the multi-operand multiplication in the definition of \(u_1\). We would need to break this and create a tree of 2-operand multipliers. In case of multi-operand addition we do not modify since we can create multioperand addition in our code generation step. If the output of the generator was software then similar modification would have been needed to change the multi-operand addition to a binary tree of two-operand additions. Figure 4.13 demonstrates the correct form of these expression trees. Algorithm 4.2.1 demonstrates the steps required to achieve this modification to the expression trees. Each node of the tree is represented by a tuple \((op, op_1, op_2, \ldots, op_n)\) where the first element is the operation defined by that node and \(op_1, op_2, \ldots\) are the operands of that operation. Operations are defined as \(op \in \{+', '-', '*', '=', '#'\}\) corresponding to addition, negation, multiplication, assignment, and delay operations. \(n\) is the number of operands required for the current operation which can vary. Each operand can be either an integer value, a variable, or another tuple (tree structure). Functions push and shift are stack and list operators to add to the end and remove an operand.
from the beginning of the tuple. The algorithm checks the first element of the

tuple $T[0]$ for different operations. For multiplication (lines 5..12) the algorithm
breaks up multi-operand multiplication coming from the parser into a tree of
2-operand multiplications. Each operand is recursively fixed as well. For the
addition and subtraction (lines 13..22) we only remove the tuple of the form
($'*'$, $op_1$) which is not required since no actual computation is required. In case
of negation (\texttt{'-'}, \texttt{op}) or multi-operand addition (\texttt{'+'}, \texttt{op}_1, \ldots, \texttt{op}_n) we return the tuple with the operands recursively fixed. For the assignment operation (lines 23..24) we only try to fix the right hand side since the left-hand side is a variable. For buffer (delay) operation we return the tuple intact since according to the grammar we can only have a variable on the right-hand side. This algorithm is applied to each polynomial (assignment in the grammar).

### 4.2.3 Size Calculator

To calculate the size of the variables and the nodes in the expression tree involved in the evaluation of the polynomials we use interval arithmetic [BO97]. Unlike the design of simple operators where the output and the size are easily calculated, complex expressions in the input polynomials would require interval arithmetic to find the range of each variable involved. To increase the speed of evaluation, we use redundant representation throughout (section 4.1) and we have to take this into consideration when we calculate the sizes. This is to avoid unnecessary representation of a variable or a constant in redundant form where the conventional form suffices. As mentioned before, the output of an expression can have smaller size than if we rely on a naïve implementation where we keep increasing the size of the output to accommodate the additions and multiplication. In this case we would need to truncate the variable to the required bit-width to lower the cost of the implementation. We showed the required computation bit-width according to these sizes in section 4.1.4 and in the case of redundant variables we proved the required operation in theorem 4.1.1. According to interval arithmetic, the ranges of the result of addition, subtraction, and multiplication of two operators with
ranges \([min_1, max_1]\) and \([min_2, max_2]\) are:

\[
[min_1, max_1] + [min_2, max_2] = [min_1 + min_2, max_1 + max_2]
\]

\[
[min_1, max_1] - [min_2, max_2] = [min_1 - max_2, max_1 - min_2]
\]

\[
[min_1, max_1] \times [min_2, max_2] = [\min\{min_1 \times min_2, min_1 \times max_2, max_1 \times min_2, max_1 \times max_2\},\max\{min_1 \times min_2, min_1 \times max_2, max_1 \times min_2, max_1 \times max_2\}]
\]

For negation we can use subtraction with the first operator in the range \([0, 0]\):

\[
- [min, max] = [\neg max, \neg min]
\]

We repeat the example from section 4.1.4. If two variables \(a\) and \(b\) represented as 4-bit 2’s complement numbers are in the range \([-8, 7]\), we have:

\[
a \times b = [-8, 7] \times [-8, 7] = [-56, 64]
\]

A variable with the range \([-56, 64]\) requires 8 bits in 2’s complement representation. However \(-a \times b\) is in range \([-64, 56]\) and only requires 7 bits. Another example is when we use the recoders to go to radix-4 we have to extend 1 bit position for odd number of bits in multiplier. If the multiplier is in CS form then the recoder also generates an extra row, so overall we would have extra 2 bits of output which are not necessary in the rest of evaluation. Algorithm 4.2.2 demonstrates the steps required to calculate the sizes of variables in the expression trees. Each node of the tree is represented by a tuple \((op, op_1, op_2, \ldots, op_n)\) as before. The first element is the operation defined by that node and \(op_1, op_2, \ldots\) are the operands of that operation. The algorithm updates the set of tuples of the form \((T, size, min, max)\) where \(T\) represents the node in the expression tree, \(size\) represents the size of that node (number of bits). A negative value for \(size\) indicates
that the node is represented in redundant form (Carry-Save) and a positive value indicates a conventional representation. The node range is returned as \([\text{min}, \text{max}]\). Since we traverse the tree to find the sizes of each node, we need this range to use in the interval arithmetic (\textit{size} alone is not enough). In any invocation of this algorithm, if the size of any of the operands is not defined (\(\text{op}_i, 0, -\infty, \infty\)), we cannot calculate the size of the current node of the tree. This means the current polynomial relies on a definition of another polynomial, hence we would have to continue calling this algorithm with other polynomials until we have the sizes of all variables (algorithm 4.2.3). We calculate the constant (coefficient) sizes for 2’s complement representation as:

\[
\text{const-size}(T) = \begin{cases} 
\lceil \log_2(T + 1) \rceil + 1, & \text{if } T \geq 0 \\
\lceil \log_2|T| \rceil + 1, & \text{otherwise}
\end{cases}
\]

In the case where we have reached the tree terminal (\(<\text{id}>\) or \(<\text{digit}>\)) we either have an input variable or a calculated size in \(S\) (lines 2..4) or a constant/coefficient (lines 4..5). In both cases we have the size and we can return it otherwise the size of the variable (\(u_1, \ldots\)) is not defined yet and we should wait (lines 6..7) for other polynomials to go through this size calculations. In the case of multi-operand addition (lines 10..21) we have used the interval arithmetic to find the size required for the evaluation of this node. In the case where the size of one of the operands cannot be determined, we would have to wait for the next time we run this algorithm on this node (lines 15..17). The size of the returned tuple (line 21) is always negative, since the output of multi-operand addition is always in redundant form \((- \max\{\text{const-size}(\text{min}_{ret}), \text{const-size}(\text{max}_{ret})\})\). The absolute value of the size is the maximum of the number of bits we require for the range of the output (\(\text{min}_{ret}, \text{max}_{ret}\)). At this stage we only have the operator ’−’ for
Algorithm 4.2.2 VAR-SIZE(T, S)

Input: T = (op, op₁, op₂, ..., opₙ)
Input: S = \{(T₁, size₁, min₁, max₁), ...\}
Output: S = \{(T₁, size₁, min₁, max₁), ...

1: if T ≠ tuple then
2: if T ∈ nodes(S) then
3: return S[T]
4: else if T ∈ Z then
5: return (T, const-size(T), T, T)
6: else
7: return (T, 0, -∞, ∞)
8: end if
9: end if
10: if T[0] = '+' then
11: min_ret ← 0
12: max_ret ← 0
13: for all opᵢ ∈ \{op₁, op₂, ..., opₙ\} do
14: (Tᵢₚₑₙₐₓ, size, min, max) ← VAR-SIZE(opᵢ)
15: if size = 0 then
16: return (T, 0, -∞, ∞)
17: end if
18: min_ret ← min_ret + min
19: max_ret ← max_ret + max
20: end for
21: return (T, - max{const-size(min_ret), const-size(max_ret)}, min_ret, max_ret)
22: else if T[0] = '-' then
23: (Tᵢₚₑₙₐₓ, size, min, max) ← VAR-SIZE(op₁)
24: if size = 0 then
25: return (T, 0, -∞, ∞)
26: end if
27: return (T, - max{const-size(-max), const-size(-min)}, -max, -min)
28: else if T[0] = '*' then
29: (T₁, size₁, min₁, max₁) ← VAR-SIZE(op₁)
30: (T₂, size₂, min₂, max₂) ← VAR-SIZE(op₂)
31: if size₁ = 0 ∨ size₂ = 0 then
32: return (T, 0, -∞, ∞)
33: end if
34: min_ret ← min\{min₁ × min₂, min₁ × max₂, max₁ × min₂, max₁ × max₂\}
35: max_ret ← max\{min₁ × min₂, min₁ × max₂, max₁ × min₂, max₁ × max₂\}
36: if size₂ > 0 ∧ op₁ ∈ Z ∧ op₁ = 2ᵏ then
37: sign ← 1
38: else
39: sign ← -1
40: end if
41: return (T, sign × max\{const-size(min_ret), const-size(max_ret)\}, min_ret, max_ret)
42: end if
Algorithm 4.2.2 VAR-SIZE(T, S) (continued)

43: if T[0] = '=' then
44: if op_1 \in nodes(S) then
45: \hspace{1em} return S[op_1]
46: end if
47: (T_{imp}, size, min, max) \leftarrow VAR-SIZE(op_2)
48: if size \neq 0 then
49: S \leftarrow S \cup \{(op_1, size, min, max)\}
50: end if
51: return (op_1, size, min, max)
52: else if T[0] = '#' then
53: return VAR-SIZE(op_1)
54: end if

negation and as describe before we calculate the size requirement and the range using interval arithmetic (lines 22..27). In the case of multiplication (lines 28..41), we use interval arithmetic as explained before to find the size and range of this node by using the ranges of op_1 and op_2 (lines 34..35). The only consideration here is the sign (representation) of the current node. In all cases we have a redundant representation for this node (negative sign) except (lines 36..37) for the case where we multiply a variable represented as 2’complement (size_{2} > 0) by a multiple of 2 (op_1 \in \mathbb{Z} \land op_1 = 2^k). In this case we will have a simple shift to the left and the representation can stay in 2’ complement form (sign > 0).

We should note that in the case where the second operand is redundant and we multiply by multiples of 2 we still do a simple shift however, the sign would be negative since we are left-shifting a CS operand. In the case of an assignment (lines 43..51) we first calculate the size of the right-hand side of the assignment and if we get a valid size we create a tuple for the op_1 in the set S (line 49). This is the definition of the variable represented by op_1. The returned tuple goes to the algorithm 4.2.3 so that we can make sure we have found the definition (sizes) for all polynomials. op_1 in this case represents the output of the system or the variables defined (\{u_1, u_2, \ldots\}) during the common subexpression and optimization steps (section 3). In the case of buffer operation, we return the same size and range.
since this operation does not modify them (lines 52..53). Algorithm 4.2.3 continues calculating the sizes until we have found the definition for all variables involved. $S$ represents the tuples as defined before with the sizes where we populate with the inputs which are known (line 1). In this case we assumed our inputs $(x_1, \ldots)$ are represented as 8-bit 2’complement signed integers. When this algorithm returns we have the sizes we need for implementing all the variables in the next phase. Going back to the example used in the previous phase (set of polynomials \{i = 

\[u_1-u_2+u_3; u_1 = a^2_2; u_2 = 5a_1a_2b_1; u3 = \#u_1; \}}\), considering the inputs $a_2, b_2, a_1, b_1$ as 8-bit 2’complement signed integers after running this phase we will have sizes and ranges presented in table 4.6. The representation is displayed based on the sign of the returned size for the corresponding variable where negative value is a redundant form (Carry-Save) and positive values is a 2’s complement form as explained before. If we do not use interval arithmetic then for example in the case of $i$ since we do not know the ranges of $u_1, u_2,$ and $u_3$ and we rely on their sizes, the 3-operand addition would result in 26-bit output for $i$ where in reality we only need 25-bits (CS) according to the range we have calculated for $i$.

**Algorithm 4.2.3** POLY-SIZE($T$)

**Input:** $T = \{T_1, \ldots T_n\}$

**Input:** $T_i = (op, op_1, op_2, \ldots, op_n)$

**Output:** $S = \{(T_1, \text{size}_1, \text{min}_1, \text{max}_1), \ldots\}$

1: $S \leftarrow \{(x_1, 8, -128, 127), (x_2, 8, -128, 127), \ldots\}$
2: $\text{continue} \leftarrow \text{True}$
3: while $\text{continue}$ do
4:   $\text{continue} \leftarrow \text{False}$
5:   for all $T_i \in T$ do
6:     $(T_{\text{tmp}}, \text{size}, \text{min}, \text{max}) \leftarrow \text{VAR-SIZE}(T_i, S)$
7:     if $\text{size} = 0$ then
8:       $\text{continue} \leftarrow \text{True}$
9:     end if
10:   end for
11: end while
12: return $S$
<table>
<thead>
<tr>
<th>Var</th>
<th>Representation</th>
<th>Size</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_2$</td>
<td>2’s</td>
<td>8</td>
<td>−128</td>
<td>127</td>
</tr>
<tr>
<td>$b_2$</td>
<td>2’s</td>
<td>8</td>
<td>−128</td>
<td>127</td>
</tr>
<tr>
<td>$a_1$</td>
<td>2’s</td>
<td>8</td>
<td>−128</td>
<td>127</td>
</tr>
<tr>
<td>$b_1$</td>
<td>2’s</td>
<td>8</td>
<td>−128</td>
<td>127</td>
</tr>
<tr>
<td>$u_1$</td>
<td>CS</td>
<td>16</td>
<td>−16256</td>
<td>16384</td>
</tr>
<tr>
<td>$u_2$</td>
<td>CS</td>
<td>25</td>
<td>−10485760</td>
<td>10403840</td>
</tr>
<tr>
<td>$u_3$</td>
<td>CS</td>
<td>16</td>
<td>−16256</td>
<td>16384</td>
</tr>
<tr>
<td>$i$</td>
<td>CS</td>
<td>25</td>
<td>−10436352</td>
<td>10518528</td>
</tr>
</tbody>
</table>

Table 4.6: Variable Sizes and Ranges for $\{i = u_1 - u_2 + u_3; u_1 = a_2^2; u_2 = 5a_1a_2b_1; u_3 = #u_1;\}$

4.2.4 Code Generator

At this point we know the size of each node where we try to generate the code for. The nodes in the expression tree are tuples of the form $(op, op_1, op_2, \ldots, op_n)$ as before. We generate Verilog code, however this can be modified and adapted to any platform. At each node once we generate the code, we return the resulting variable(s) from Verilog code, the size, possible negation, and the arrival time as $(var, size, neg, time)$. This means we have to generate the code for the operands first. We have included the Verilog code of building blocks in section A.3. Following we discuss each operation:

4.2.4.1 Multi-Operand Addition

As explained before, we want to generate the code for a node of the form $‘+’, op_1, op_2, \ldots, op_n)$. We have already explained how we handle the adder tree in section 4.1.1. The only considerations here are the negative sign of an operand and the arrival time. The negation of an operand would result in logical negation of the variable associated with the operand in Verilog with addition of 1 to the multi-operand addition. In the case where the operand is in CS form, we would have to apply this negation to both vectors resulting in addition of 2 to the multi-operand addition. The solution in [PZ91, PZ93] does not offer any improvements
here, since our adder block \([4 : 2]\) has similar delays for each input and output and we have the operands arriving at different times. Instead we prioritize the operands based on the arrival time to decrease the overall delay. For example, if there are 3 operands in CS form are added, and one of the operands arrives later more than \(1.5\tau_{FA}\) after other operands, if we schedule the addition for this operand first, our overall delay becomes \(2 \times 1.5\tau_{FA}\). Whereas, if we schedule the other two operands first the total delay is only \(1.5\tau_{FA}\), since the result of addition of other two CS operands is ready before the last \([4 : 2]\) addition. Greedy Algorithm 4.2.4 shows the heuristics we have used to address the arrival time improvement. The

<table>
<thead>
<tr>
<th>Algorithm 4.2.4 SCHEDULE(T)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> ( T = (op_1, op_2, \ldots, op_n)</td>
</tr>
<tr>
<td><strong>Output:</strong> ( G = {(var, size, neg, time)} )</td>
</tr>
<tr>
<td>1. ( G \leftarrow T )</td>
</tr>
<tr>
<td>2. \textbf{repeat}</td>
</tr>
<tr>
<td>3. ( G \leftarrow \text{SORT}(G) )</td>
</tr>
<tr>
<td>4. \textbf{if} ( G[4][\text{time}] - G[3][\text{time}] &gt; \tau_{FA} ) \textbf{then}</td>
</tr>
<tr>
<td>5. ( \text{push}(G, \text{GENERATE-3 : 2}(\text{shift}(G), \text{shift}(G), \text{shift}(G))) )</td>
</tr>
<tr>
<td>6. \textbf{else}</td>
</tr>
<tr>
<td>7. ( \text{push}(G, \text{GENERATE-4 : 2}(\text{shift}(G), \text{shift}(G), \text{shift}(G), \text{shift}(G))) )</td>
</tr>
<tr>
<td>8. \textbf{end if}</td>
</tr>
<tr>
<td>9. \textbf{until} (</td>
</tr>
<tr>
<td>10. \textbf{return} ( G )</td>
</tr>
</tbody>
</table>

Algorithm sorts the operands based on the arrival time (line 3) and in the case where it’s more suitable to schedule a \([3 : 2]\) adder we generate a \([3 : 2]\) adder (lines 4..5). Otherwise we generate a \([4 : 2]\) adder and in either case we push back the two resulting operands in the list of operands. The generation creates the required variables, sign extend the operands, negate some operands and create the blocks required. This continues until we are left with two operands (CS). For example the output for the polynomial \(i = a_2 + b_2 - a_1\) is as following:

```
reg signed [7:0] a2;
reg signed [7:0] b2;
reg signed [7:0] a1;
// Adder Tree with 4 operands
```
The multi-operand adder has 4 operands, since we have a 2’s complement negation which results in bit-negation of \(a_1\) and addition of 1.

### 4.2.4.2 Multiplication

As explained before, to reduce the number of multiples of the multiplicand, we recode the multiplier from CS/2’s complement representation to minimally redundant radix-4 digit-set (\(\{\overline{2}, \overline{1}, 0, 1, 2\}\)). Once we have generated the multiples, we use multi-operand addition as before to calculate the result. As mentioned before, the multi-operand adder, uses a tree of redundant adders to reduce the partial products to 2 vectors representing the carry-save representation of the result. The truncation again can happen in the case we have sign extended the multiplier (odd number of digits) and the extra row for CS to radix-4 decoder. Following is the Verilog code generated for \(i = a_2 \times b_2\). We have only presented the multiple generation code for the first row due to space:

```verilog
wire signed [10:0] tmp0;
assign tmp0 = {{4{a2[7]}},{a2[6:0]});
wire signed [10:0] tmp1;
assign tmp1 = {{4{b2[7]}},{b2[6:0]});
// Negate
wire signed [7:0] tmp2;
assign tmp2 = ~a1;
wire signed [10:0] tmp3;
assign tmp3 = {{4{tmp2[7]}},{tmp2[6:0]});
wire signed [9:0] tmp4_s;
wire signed [9:0] tmp4_c;
FourTwoFixed #( .WIDTH(10) ) adder4_4 ( .a(tmp0) , .b(tmp1) , .c(tmp3) , .d( 'b00000000001) , .sum(tmp4_s) , .carry(tmp4_c) );
wire signed [9:0] i_s;
wire signed [9:0] i_c;
assign i_s = tmp4_s;
assign i_c = tmp4_c;
```
wire [3:0] tmp_sgn0;
wire [3:0] tmp_p10;
wire [3:0] tmp_p20;
TC_Recoder #( .WIDTH(8) ) reccoder_0 (.m(b2) , .sgn(tmp_sgn0) , .p1(tmp_p10) , .p2(tmp_p20));

// Generate multiples \{-2,-1,1,2\}
wire signed [8:0] tmp1;
wire signed [8:0] tmp2;
NotE #( .WIDTH(9) ) not_2 (.inp(tmp1) , .inv(tmp_sgn0[0]) , .out(tmp2));
wire signed [8:0] tmp3;
assign tmp3 = ...;
wire signed [8:0] tmp4;
NotE #( .WIDTH(9) ) not_4 (.inp(tmp3) , .inv(tmp_sgn0[1]) , .out(tmp4));
wire signed [10:0] tmp5;
assign tmp5 = ...;
wire signed [8:0] tmp7;
NotE #( .WIDTH(9) ) not_7 (.inp(tmp6) , .inv(tmp_sgn0[2]) , .out(tmp7));
wire signed [12:0] tmp8;
assign tmp8 = ...;
wire signed [8:0] tmp10;
NotE #( .WIDTH(9) ) not_10 (.inp(tmp9) , .inv(tmp_sgn0[3]) , .out(tmp10));
wire signed [14:0] tmp11;
assign tmp11 = {{tmp10[8:0]} , {6[1'b0]}};
wire signed [7:0] tmp12;
assign tmp12 = {1'b0, {tmp_sgn0[3]}, 1'b0, {tmp_sgn0[2]}, 1'b0, {tmp_sgn0[1]}, 1'b0, {tmp_sgn0[0]}};

// Adder Tree with 5 operands
wire signed [16:0] tmp13;
assign tmp13 = {{9{tmp2[8]}}, {tmp2[7:0]});
wire signed [16:0] tmp14;
assign tmp14 = {{7{tmp5[10]}}, {tmp5[9:0]});
wire signed [16:0] tmp15;
assign tmp15 = {{5{tmp8[12]}}, {tmp8[11:0]});
wire signed [16:0] tmp16_s;
wire signed [16:0] tmp16_c;
ThreeTwo #( .WIDTH(17) ) adder3_16 (.a(tmp13) , .b(tmp14) , .c(tmp15) , .sum(tmp16_s) , .carry(tmp16_c));

84
wire signed [16:0] tmp17;
assign tmp17 = \{\{3{tmp11[14]}\}\},\{tmp11[13:0]\});
wire signed [16:0] tmp18;
assign tmp18 = \{\{10{tmp12[7]}\}\},\{tmp12[6:0]\});
wire signed [15:0] tmp19_s;
wire signed [15:0] tmp19_c;
FourTwoFixed #(.WIDTH(16) ) adder4_19 ( .a (tmp17) , .b(tmp18) , .c (tmp16_s) , .d(tmp16_c) , .sum(tmp19_s) , .carry (tmp19_c) );
wire signed [15:0] i_s;
wire signed [15:0] i_c;
assign i_s = tmp19_s;
assign i_c = tmp19_c;

4.2.4.3 Multiplication By Constant

Multiplication by a constant is simplified form of multiplication. We represent the 
constant in the canonical form to minimize the number of non-zero bits [EL03] and 
generate the multiples of the multiplicand. Similarly we pass on the rows to the 
multi-operand addition logic to generate the result. The only consideration here is 
that when we multiply by multiples of 2, we only need to left-shift the operands. 
In the case of CS multiplicand, we would have to left-shift both vectors. The 
technique used for multiple-constant multiplication in [EL03, PSC96] by exposing 
common subexpressions among the constant multipliers can also reduce the size. 
Following we present the Verilog code generated for \( i = 7a_2 \):
wire signed [10:0] tmp3_c;
ThreeTwoTruncate #(WIDTH(11)) adder3_3 (.a(tmp0) , .b(tmp1) , .c(tmp2) , .sum(tmp3_s) ,
.carry(tmp3_c));
wire signed [10:0] i_s;
wire signed [10:0] i_c;
assign i_s = tmp3_s;
assign i_c = tmp3_c;

4.2.4.4 Assignment

This is a simple assignment of the variable(s) generated on the right-hand side to the variable on the left-hand side and matching the sizes and representation. Last 4 lines of the example above for \( i = 7a_2 \) demonstrates this assignment. In this case the synthesis tool would ignore this assignment by replacing \( i_s \) and \( i_c \) with corresponding right-hand side variables \( tmp3_s \) and \( tmp3_c \). This is required since when the code is generated on the right-hand side we do not know that it’s part of an assignment (depth first traversal of expression tree). Additionally this assignment is required, since we could be using the variable defined in the left-hand side in other polynomials.

4.2.4.5 Delay

The code for this operation is similar to the assignment. The left-hand side is defined as a new variable (reg) in Verilog with matching size to the right-hand side. However the assignment is different since we need a delay element to pass on a product to the next clock cycle. Following shows the part of the code generated for \( \{ i = a_2 \times b_2; q = \#i \} \):

reg signed [7:0] a2;
reg signed [7:0] b2;
...
wire signed [15:0] i_s;
wire signed [15:0] i_c;
assign i_s = tmp19_s;
assign i_c = tmp19_c;
reg signed [15:0] q_s;
reg signed [15:0] q_c;
...
always @ (posedge clk)
begin : BUFFERS
    q_s <= i_s;
    q_c <= i_c;
end
...

Due to the space limitation, presenting the Verilog code for our sample sets of polynomials is not feasible. However we present the results of the synthesis of our optimized implementation using our generator.

4.3 Results

We applied the implementation techniques mentioned to the set of polynomials resulted by the expansion of the Volterra series of a non-linear system. We present this application in section 5. The set of polynomials \( \{y_I, y_Q\} \) defined over \( \mathbb{Z}[x_I, x_{I[1]}, x_Q, x_{Q[1]}] \) is listed in equations 5.5 and 5.6 in section 5. The polynomials in the critical path are:

\[
\begin{align*}
    u_{30} &= x_I^2 \\
    u_9 &= x_I^2 + x_Q^2 \\
    u_{25} &= 30u_9 \\
    u_1 &= x_I u_{14} + x_{I[1]} u_{10} + x_{Q[1]} u_{15} + u_{25} + 5783552 \\
    y_I &= x_I u_1 + x_Q u_3 + x_{I[1]} u_5 + x_{Q[1]} u_7 + 128u_6 - 2097152
\end{align*}
\]

with the sizes listed in Table 4.7. Ignoring the logic invert delays for the subtraction and considering delay of multiplexer as an XOR delay, the delay are as
<table>
<thead>
<tr>
<th>Var</th>
<th>Representation</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_I$</td>
<td>2's</td>
<td>8</td>
</tr>
<tr>
<td>$x_I[1]$</td>
<td>2's</td>
<td>8</td>
</tr>
<tr>
<td>$x_Q$</td>
<td>2's</td>
<td>8</td>
</tr>
<tr>
<td>$x_Q[1]$</td>
<td>2's</td>
<td>8</td>
</tr>
<tr>
<td>$u_{30}$</td>
<td>CS</td>
<td>16</td>
</tr>
<tr>
<td>$u_{31}$</td>
<td>CS</td>
<td>16</td>
</tr>
<tr>
<td>$u_9$</td>
<td>CS</td>
<td>17</td>
</tr>
<tr>
<td>$u_{25}$</td>
<td>CS</td>
<td>21</td>
</tr>
<tr>
<td>$u_{14}$</td>
<td>CS</td>
<td>14</td>
</tr>
<tr>
<td>$u_{10}$</td>
<td>CS</td>
<td>14</td>
</tr>
<tr>
<td>$u_{15}$</td>
<td>CS</td>
<td>13</td>
</tr>
<tr>
<td>$u_1$</td>
<td>CS</td>
<td>25</td>
</tr>
<tr>
<td>$u_3$</td>
<td>CS</td>
<td>22</td>
</tr>
<tr>
<td>$u_5$</td>
<td>CS</td>
<td>23</td>
</tr>
<tr>
<td>$u_7$</td>
<td>CS</td>
<td>18</td>
</tr>
<tr>
<td>$u_6$</td>
<td>CS</td>
<td>17</td>
</tr>
<tr>
<td>$y_I$</td>
<td>CS</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 4.7: Critical Path Variables Sizes

following:

$$
\tau_{u_{30}} = \tau_{2S-RECODER} + \tau_{PP-GEN} + \tau_{adder}(5) \\
= 0.5\tau_{FA} + \tau_{FA} + 2.5\tau_{FA} = 4\tau_{FA}
$$

where the $x_I$ is recoded from 2’s complement to minimally redundant radix-4 representation. $\tau_{adder}(5)$ comes from the partial product rows generated (the extra row is for the negative multiples).

$$
\tau_{u_{9}} = \tau_{u_{30}} + \tau_{adder}(4) \\
= 4\tau_{FA} + 1.5\tau_{FA} = 5.5\tau_{FA}
$$
where we have addition of two operands in CS form.

\[
\mu_{25} = \mu_9 + \tau_{ADDER}(5)
\]

\[
= 5.5\tau_{FA} + 2.5\tau_{FA} = 8\tau_{FA}
\]

where the constant multiplier $30 \times$ has 5 rows since $30 = 1000\bar{1}0$ in canonical form and operand $u_9$ is in CS form (the extra line is for the negative multiple).

\[
\mu_{1} = \mu_{25} + \tau_{CS-RECORDER} + \tau_{PP-GEN} + \tau_{ADDER}(8) + \tau_{ADDER}(9)
\]

\[
= 8\tau_{FA} + 1.5\tau_{FA} + \tau_{FA} + 3\tau_{FA} + 3.5\tau_{FA} = 17\tau_{FA}
\]

where the $\tau_{ADDER}(8)$ is due to the 8 rows of multiples generated after recoding $u_{10}$ and $\tau_{ADDER}(9)$ comes from the definition of $u_1$ and the representation of the operands.

\[
\mu_{I} = \mu_{1} + \tau_{CS-RECORDER} + \tau_{PP-GEN} + \tau_{ADDER}(14) + \tau_{ADDER}(11)
\]

\[
= 17\tau_{FA} + 1.5\tau_{FA} + \tau_{FA} + 4.5\tau_{FA} + 4\tau_{FA} = 28\tau_{FA}
\]

where the $\tau_{ADDER}(14)$ is due to the 14 rows of multiples generated after sign extending and recoding $u_1$ and $\tau_{ADDER}(11)$ comes from the definition of $u_1$ with operands in CS form. The adder tree delays are calculated from the Figure 4.2. Since $y_I$ is in CS form, we need a 32-bit adder to change to conventional form. Hence, the final critical path delay becomes:

\[
\tau_{CP} = 28\tau_{FA} + \tau_{ADDER-32bit}
\]

We synthesized these approaches for Altera Cyclone V 5CEFA7F31C7 FPGA using Quartus 13 and simulated using Multisim. Table 4.8 lists out the number of DSP’s and adaptive logic modules (ALM) required for each implementation.
Our implementation does not rely on the hard-cores (DSP) available on FPGA’s. The first two rows are the result of the synthesis of the polynomials optimized with our high algebraic method (Chapter 3). First row reports the synthesis with the constraint not to use DSP blocks. We perform this synthesis to have a fair comparison in terms of cost and speed with our implementation in which we use redundant arithmetic. We saw an increase in the cost of implementation due to redundant representation of our variable. Table 4.9 shows the maximum frequency of each implementation. We see that even though our approach does not use the DSP’s on the FPGA, because of using the redundant arithmetic, we still see an improvement in the speed in comparison with the optimized polynomial using DSP’s. According to [alt07] the digital pre-distorters have a bandwith of 15 to 20 MHz, which puts our solution ahead in terms of speed.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>DSP (%)</th>
<th>ALM (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poly. Optimization (no DSP)</td>
<td>0(0%)</td>
<td>1739(3%)</td>
</tr>
<tr>
<td>Poly. Optimization</td>
<td>22(14%)</td>
<td>274(&lt; 1%)</td>
</tr>
<tr>
<td>Poly. + Arith. Optimization</td>
<td>0(0%)</td>
<td>4964(9%)</td>
</tr>
</tbody>
</table>

Table 4.8: FPGA Implementation Costs

<table>
<thead>
<tr>
<th>Implementation</th>
<th>$f_{Max}$ (MHz)</th>
<th>Improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poly. Optimization (no DSP)</td>
<td>17.6</td>
<td>+44.3%</td>
</tr>
<tr>
<td>Poly. Optimization</td>
<td>25.4</td>
<td></td>
</tr>
<tr>
<td>Poly. + Arith. Optimization</td>
<td>30.4</td>
<td>+72.7%</td>
</tr>
</tbody>
</table>

Table 4.9: FPGA Implementation Maximum Frequency

### 4.4 Summary

After application of our algebraic method in previous chapter, we focused on hardware implementation of Volterra polynomials. We chose redundant representation, to speed up the evaluation. This is due to the fact that unlike conventional adders, redundant adder delay does not rely on precision of the operands. We presented
the components such as multi-operand addition, redundant multiplication using recoders, constant multiplication, and carry-save truncation. To avoid manual coding of such systems, we presented a generator that we have developed. Our generator parses the set of input polynomials and generates Verilog HDL code. To do so, the generator creates expression trees, modifies them, and using interval arithmetic calculates the required precision of each node of the expression trees. The generator also considers the arrival time of operands in multi-operand addition to schedule the additions to lower the overall delay. Our results show that even though our implementation, using redundant arithmetic, competes with the DSP blocks available on FPGA’s, our implementation outperforms the design where we only apply our algebraic method.
CHAPTER 5

Application

Polynomials can be used to model linear and non-linear systems. These polynomials relate the inputs of the system to the outputs of the system. The evaluation of the polynomials and computing the output of these systems, requires a set of additions and multiplications. Systems that have memory effect also rely on the delayed inputs and in certain systems (Infinite Impulse Response filters) on the delayed outputs. Hardware implementation of such systems would need to address the issues such as calculation precision and area/latency/power requirements. We focus on nonlinear systems with memory which can be modeled with Volterra polynomials. The methods we presented in previous chapters can be applied to optimize and implement Volterra polynomials. Previous attempts to implement such systems, relied on hand-optimization and lacked any automatic algebraic optimization. We will present an application where Volterra polynomials can be used to model the nonlinear system in section 5.2. Our techniques and tools can be equally applied to any nonlinear system modeled by Volterra series.

5.1 Volterra Series

In linear systems the output \((y(t))\) of the system is calculated by the convolution of the input \((x(t))\) and the impulse response of the system \((h(t))\):

\[
y(t) = x(t) * h(t) = \int_{-\infty}^{\infty} h(\tau)x(t - \tau)d\tau
\]
and in the case of discrete systems [Pro07]:

\[ y(n) = x(n) \ast h(n) = \sum_{k=-\infty}^{\infty} h(k)x(n - k) \]

In causal systems where the current value of output only relies on the current and previous values of the input the integration (summation in the case of discrete system) can be calculated from 0 to infinity. In real implementations, the amount of memory of the system is limited in the summation of the discrete system:

\[ y(n) = \sum_{k=0}^{N-1} h(k)x(n - k) \quad (5.1) \]

This type of filter is called Finite-Impulse Response (FIR) [Pro07]. In many systems the behavior of the system cannot be modeled with a linear model. Volterra series is an extension of linear convolution integral [MS00]:

\[
\begin{align*}
y(t) &= h_0 + \int_{-\infty}^{\infty} h_1(\tau_1)x(t - \tau_1)d\tau_1 \\
&\quad + \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} h_2(\tau_1, \tau_2)x(t - \tau_1)x(t - \tau_2)d\tau_1d\tau_2 \\
&\quad \vdots \\
&\quad + \int_{-\infty}^{\infty} \ldots \int_{-\infty}^{\infty} h_n(\tau_1, \ldots, \tau_n)x(t - \tau_1) \ldots x(t - \tau_n)d\tau_1d\tau_2 \ldots d\tau_n \\
&\quad \vdots
\end{align*}
\]

This can be seen as infinite sum of higher order impulse response convolved with the interactions of the inputs [MML01]. \( h_1 \) is the linear impulse response of the
system. The discrete form of this model is:

\[
y(n) = h_0 + \sum_{k_1=-\infty}^{\infty} h_1(k_1)x(n - k_1) + \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} h_2(k_1, k_2)x(n - k_1)x(n - k_2) + \cdots
\]

\[
+ \sum_{k_1=-\infty}^{\infty} \cdots \sum_{k_n=-\infty}^{\infty} h_n(k_1, k_2, \ldots, k_n)x(n - k_1) \ldots x(n - k_n)
\]

\[h_n\] is called the \(n^{th}\) order Volterra kernel. To implement filters using this model, we use truncated Volterra series up to the degree that we are trying to model. Also the memory of the system in each summation can be limited to make the design feasible. The causal discrete nonlinear system of degree \(M\) then can be modeled as:

\[
y(n) = h_0 + \sum_{n=1}^{M} \sum_{k_1=0}^{N-1} \cdots \sum_{k_n=0}^{N-1} h_n(k_1, k_2, \ldots, k_n)x(n - k_1) \ldots x(n - k_n)
\]

where \(N - 1\) is the memory length of each kernel \((h_1, h_2, \ldots, h_M)\), even though each kernel \(h_i\) can have different memory size it’s safe to consider a fixed memory length for all kernels. For \(M = 1\) this becomes the linear system and the formula becomes the FIR filter defined in equation 5.1. The number of parameters is:

\[
P = \sum_{n=1}^{M} N^n
\]

For example for 3\(^{rd}\) degree Volterra series with one input and 2 memory location we have \(P = 39\) parameters. To reduce this number, due to symmetry of Volterra
kernels, triangular form of the Volterra model is used [MS00]:

\[ y(n) = h_0 + \sum_{n=1}^{M} \sum_{k_1=0}^{N-1} \sum_{k_2=k_1}^{N-1} \cdots \sum_{k_n=k_{n-1}}^{N-1} h_n(k_1, k_2, \ldots, k_n)x(n-k_1)\ldots x(n-k_n) \]

The number of parameters with this representation becomes [TSM09]:

\[ P_{sym} = \sum_{n=1}^{M} \frac{(N - 1 + n)!}{(N - 1)!n!} \]

For example for 3\textsuperscript{rd} degree Volterra series with one input and 2 memory location we have \( P_{sym} = 19 \) parameters. For multiple input single output (MISO) systems with \( k \) inputs \( (x_1, x_2, \ldots, x_k) \) triangular Volterra model becomes [THS02b, THS03]:

\[ y(n) = h_0 + \sum_{n=1}^{M} \sum_{i_1=1}^{k} \sum_{i_2=1}^{k} \sum_{i_3=1}^{N-1} \sum_{k_1=0}^{N-1} \cdots \sum_{k_n=k_{n-1}}^{N-1} h_{i_1, \ldots, i_n}(k_1, k_2, \ldots, k_n)x_{i_1}(n-k_1)\ldots x_{i_n}(n-k_n) \] (5.2)

The kernels \( h_{i_1, \ldots, i_n} \) in this case take into consideration the interaction between the inputs. The number of parameters with this representation becomes [TSM09]:

\[ P_{multi} = \sum_{n=1}^{M} k^n \frac{(N - 1 + n)!}{(N - 1)!n!} \]

For example for 3\textsuperscript{rd} degree Volterra series with 2 inputs and 2 memory location we have \( P_{multi} = 110 \) parameters. Identification of the Volterra series has been studied a lot and there are techniques available using Linear Regression [MML01, GZ10], Orthonormal Base Function [THS02a], Polynomial Kernel Regression (SVM Regression) [Sch05, FS06, TSM09]. The purpose of this research is not to improve the identification of Volterra series or similar polynomial datapath representing a non-linear system. We present Linear Regression identification without any improvement and our methods can be applied to any Volterra polynomials regardless
of the identification method.

5.1.1 Linear Regression

Linear regression removes the assumption of Gaussian input with zero mean which is used with traditional Orthonormal Based estimation [THS02a]. It also avoids the requirement of having large samples of data to converge. Given a sample set of inputs and the corresponding measured outputs \( \{(X^{[i]}, y^{[i]}), \ldots, (X^{[S]}, y^{[S]})\} \) where \((X^{[i]}, y^{[i]}) = (x^{[i]}_1, x^{[i]}_2, \ldots, x^{[i]}_k, y^{[i]}) \) are the \( j \)-th sample output with the corresponding inputs. Linear regression tries to find parameters of a polynomial estimating the system:

\[
 f(x_1, \ldots, x_k) = \sum_{j=1}^{P} \gamma_j \varphi_j(x_1, \ldots, x_k)
\]

Where \( \gamma_j \in \mathbb{R} \) and \( \varphi_j : \mathbb{R}^k \rightarrow \mathbb{R} \) is in our case a monomial of \( x_1, \ldots, x_k \) up to order \( M \) (\( M \) is the order of the Volterra series). This is done by minimizing the mean squared error over the sample data [FS06, MS00]:

\[
 \arg\min_{\gamma} \frac{1}{S} \sum_{i=1}^{S} (f(x_1, \ldots, x_k) - y_i)^2
\]

If we rename and represent the kernel parameters as a vector:

\[
 h^T = \begin{pmatrix} \gamma_1 & \gamma_2 & \cdots & \gamma_P \end{pmatrix}
\]

and the combination of inputs associated with each \( \gamma_j \) monomial as:

\[
 x^T = \begin{pmatrix} 1 & x_1 & x_2 & \cdots & x_1^M & x_1^{M-1} x_2 & \cdots & x_k^M \end{pmatrix}
\]
The function $f$ can be represented as:

$$f = h^T x$$

For the $S$ sample data we can define the matrix $X$ as:

$$X = \begin{bmatrix}
  x^{[1]} \\
x^{[2]} \\
  \vdots \\
x^{[S]}
\end{bmatrix}$$

and based on the observed output:

$$y^T = \begin{bmatrix}
  y^{[1]} \\
y^{[2]} \\
  \vdots \\
y^{[S]}
\end{bmatrix}$$

$$y = Xh$$

To estimate the system (find $h$) using least mean square we have [LSK11, MML01]:

$$h = (X^T X)^{-1} X^T y$$

Where the number of samples $S$ has to be greater than the number of parameters $P$ and $(X^T X)^{-1} X^T$ is Moore-Penrose pseudoinverse matrix.

### 5.2 Power Amplifier Predistorter

We have presented techniques to optimize and implement Volterra polynomials so far. In this section we focus on a particular application modeled by Volterra series. Power Amplifiers (PA) are valuable components of virtually all communication applications. They are inherently non-linear due to the designs that sacrifice linearity for higher efficiency. This results in spectral growth and in-band dis-
tortions and consequently higher bit-error rate (BER). Since the performance of communication channels rely on the capacity of data, researchers have adopted a technique called baseband predistortion [LPS94]. This technique compensates for the non-linearity of the system by applying a reverse nonlinear component to the signal which causes the final amplified signal to be a linear amplification of the input signal. This makes PA a linear amplifier (Figure 5.1). The previous implementation attempts relied on hand-coding of such systems [LSK11] with very limited optimization. Digital predistortion has been the topic of research in the past decade where the predistortion happens in the digital domain by different models. To model the nonlinear system, we adopted the a Wiener-Hammerstein (W-H) model (Figure 5.2) which consists of a Linear Time Invariant (LTI) system followed by a memory-less non-linearity following by another LTI system [EP97, DZM04]. The nonlinear element acts on the input signal represented with the magnitude $r(t)$ and phase $\phi_0(t)$ (both functions of time) and generate the output represented with magnitude $A(r(t))$ and phase $\phi_0(t) + \Phi(r(t))$. This configuration is usually used to model the PA in satellite communication using traveling wave tube (TWT) or a solid-state amplifier. The amplifier works in the saturation operating region introducing nonlinearity to the output amplitude and

\[
\begin{align*}
  f(t) &= r(t)e^{j\phi_0(t)} \\
  d(t) &= A(r(t))e^{j(\phi_0(t)+\Phi(r(t)))}
\end{align*}
\]

Figure 5.2: Nonlinear Amplifier

saturation operating region introducing nonlinearity to the output amplitude and
phase of the signal. We used the following TWT model [EP97, Sal81]:

\[ A(r(t)) = \frac{\alpha_a r(t)}{1 + \beta_a r^2(t)} \] (5.3)

\[ \Phi(r(t)) = \frac{\alpha_p r^2(t)}{1 + \beta_p r^2(t)} \] (5.4)

This model represents AM-AM and AM-PM nonlinearities. We used the typical parameters (\(\alpha_a = 2.0, \beta_a = 1.0, \alpha_p = \frac{\pi}{3}, \beta_p = 1.0\)) from [PG87]. We also used two linear filters in W-H model with following parameters:

\[ T = [0.8, 0.1] \]
\[ R = [0.9, 0.2, 0.1] \]

Figure 5.3 demonstrates the effect of this nonlinear system on a random 500 QAM-64 (Quadrature Amplitude Modulation) samples. Although efforts have gone to modeling these systems using memory-less nonlinear filters or Memory polynomials [DZM04, DE12], the Volterra model is better fit for these communication systems [EP97]. In such communication systems the input is represented as In-
bound/Quadrature (IQ) signals due to easier implementation of the modulators. We trained our Volterra model using these 500 random samples and figure 5.4 demonstrates the predistorter using a Volterra model versus the most popular model which is Memory polynomials. We used 3rd order Volterra model with one memory location resulting in 4 input and 13 parameters to estimate. For the Memory polynomial model we used 5th order polynomials with 2 memory locations resulting in 6 inputs and 9 parameters. Clearly the Volterra model allows better modeling for this nonlinear system. The Memory polynomial is incapable of modeling the non-linearity completely and we do not see a clear distinction in the constellation diagram. The Volterra model is able to capture non-linearity between different degree components of the input and delayed input signals. To further simplify the circuits, researchers have proposed implementing the odd-degree memory polynomials. That implementation performs even worse since it cannot model the phase distortion [DZM04]. The parameters for the Volterra model are represented in table 5.1. Where the index of $h_{i,j}$ represents the power of the current input and delayed input represented in complex form (−1 represents complex conjugate). So far the computation was done with single-precision

(a) Volterra Model

(b) Memory Polynomials Model

Figure 5.4: Predistored Nonlinear System Output
Table 5.1: Volterra Predistorter Model Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_{0,0}$</td>
<td>$-0.0018 + 0.0042i$</td>
</tr>
<tr>
<td>$h_{1,0}$</td>
<td>$0.6893 + 0.0223i$</td>
</tr>
<tr>
<td>$h_{0,1}$</td>
<td>$-0.2429 - 0.0036i$</td>
</tr>
<tr>
<td>$h_{2,0}$</td>
<td>$-0.0007 - 0.0023i$</td>
</tr>
<tr>
<td>$h_{1,-1}$</td>
<td>$0.0003 + 0.0014i$</td>
</tr>
<tr>
<td>$h_{-1,1}$</td>
<td>$-0.0017 + 0.0001i$</td>
</tr>
<tr>
<td>$h_{0,2}$</td>
<td>$0.0022 - 0.0002i$</td>
</tr>
<tr>
<td>$h_{3,0}$</td>
<td>$0.0587 - 0.0785i$</td>
</tr>
<tr>
<td>$h_{2,1}$</td>
<td>$-0.0246 + 0.0358i$</td>
</tr>
<tr>
<td>$h_{2,-1}$</td>
<td>$-0.0134 + 0.0184i$</td>
</tr>
<tr>
<td>$h_{1,2}$</td>
<td>$0.0043 - 0.0053i$</td>
</tr>
<tr>
<td>$h_{-1,2}$</td>
<td>$0.0019 - 0.0025i$</td>
</tr>
<tr>
<td>$h_{0,3}$</td>
<td>$-0.0058 + 0.0047i$</td>
</tr>
</tbody>
</table>

floating point. The figure 5.5 compares the implementation using software floating point versus 32-bit data-path integer computation. To find the set of polynomials to work with we can expand the Volterra series and separate the real and imaginary polynomials. For example, a linear system with memory length of 1 can be

![Scatter plot](image1)

![Scatter plot](image2)

(a) Single-Precision Floating Point  
(b) 32-bit Datapath

Figure 5.5: Predistortion Implementation
expanded to (using $h_{0,0}$, $h_{1,0}$, and $h_{0,1}$ from table 5.1):

$$y = h_{0,0} + h_{1,0}x(n) + h_{0,1}x(n - 1)$$
$$= -0.0018 + 0.0042i + (0.6893 + 0.0223i)x(n)$$
$$+ (-0.2429 - 0.0036i)x(n - 1)$$
$$y_I = \Re(y) = -0.0018 + 0.6893x_I - 0.0223x_Q$$
$$- 0.2429x_{I[1]} + 0.0036x_{Q[1]}$$
$$y_Q = \Im(y) = 0.0042 + 0.6893x_Q + 0.0223x_I$$
$$- 0.2429x_{Q[1]} - 0.0036x_{I[1]}$$

To represent this system with polynomials we represent the 8-bit inputs and delayed inputs as $x_I + x_Qi$ and $x_{I[1]} + x_{Q[1]}i$. The two polynomial system representing the in-phase and quadrature predistorted signals are presented in equations 5.5 and 5.6. Since we have expanded the Volterra series with complex numbers we

$$y_I = 30x_I^3 + 40x_I^2x_Q - 19x_I^2x_{I[1]} - 9x_I^2x_{Q[1]} + 30x_Ix_Q^2$$
$$- 20x_Ix_Qx_{I[1]} - 14x_Ix_Qx_{Q[1]} + 3x_Ix_{I[1]}^2 + 2x_Ix_{I[1]}x_Q$$
$$+ x_Ix_{Q[1]}^2 + 40x_Q^3 - 5x_Q^2x_{I[1]} - 29x_Q^2x_{Q[1]} + 2x_Qx_{I[1]}^2$$
$$+ 2x_Qx_{I[1]}x_Q + 4x_Qx_{Q[1]}^2 - 3x_{I[1]}^3 - 3x_{I[1]}x_Q + 3x_{I[1]}x_{Q[1]}^2$$
$$- 3x_{Q[1]}^3 - 128x_I^2 - 128x_Ix_{I[1]} + 128x_Ix_{Q[1]} - 128x_Q^2 - 128x_Qx_{I[1]}$$
$$- 128x_{Q[1]} - 128x_{I[1]}^2 + 128x_{Q[1]}^2 + 5783552x_I - 196608x_Q$$
$$- 2031616x_{I[1]} + 3278x_{Q[1]} - 2097152$$

$$y_Q = -40x_I^3 + 30x_I^2x_Q + 29x_I^2x_{I[1]} - 5x_I^2x_{Q[1]} - 40x_Ix_Q^2$$
$$- 14x_Ix_Qx_{I[1]} + 20x_Ix_Qx_{Q[1]} - 4x_Ix_{I[1]}^2 + 2x_Ix_{I[1]}x_Q$$
$$- 2x_Ix_{Q[1]}^2 + 30x_Q^3 + 9x_Q^2x_{I[1]} - 19x_Q^2x_{Q[1]} + x_Qx_{I[1]}^2$$
$$- 2x_Qx_{I[1]}x_Q + 3x_Qx_{Q[1]}^2 + 3x_{I[1]}^3 - 3x_{I[1]}x_Q + 3x_{I[1]}x_{Q[1]}^2$$
$$- 3x_{Q[1]}^3 - 128x_I^2 + 128x_Ix_{I[1]} - 128x_Ix_{Q[1]} - 128x_Q^2 + 128x_Qx_{I[1]}$$
$$+ 128x_{Q[1]} - 196608x_I + 5783552x_Q - 32768x_{I[1]} - 2031616x_{Q[1]}$$
$$+ 4194304$$

(5.5)
have to consider the factorizations that are implied in direct evaluation [LSK11]. For example the terms $x_I^2$ and $x_I^3$ are evaluated with two multiplications instead of three. In [LSK11] hand optimization using memory elements to keep some of the products for next cycles is used to reduce the number of multiplications in Volterra series. In [LSK11] the complex products are considered for delayed nodes. This is similar to what we used in our algebraic optimization (delayed products) where we stored the value of suitable products for the upcoming cycles. The exception is that in our formulation, we do not need to only rely on current variables. It can be a mixture of all variables except for the delayed variables from the last memory locations. For example $x_I^2 x_{Q[1]}$ can be calculated for the current sample and then stored so that during the next cycle we have the value for $x_{I[1]}^2 x_{2[2]}$. This method reduces the number of multiplications by reusing the results in upcoming clock cycles.

### 5.3 Results

We have evaluated the overall implementation of two polynomials ($y_I$ and $y_Q$) using direct evaluation, direct evaluation with memory elements (buffered) [LSK11] and the algebraic method mentioned in previous chapters. Table 5.2 lists the number of additions and multiplications in each approach required to implement two polynomials $y_I$ and $y_Q$. Direct evaluation is similar to common subexpression elimination technique (CSE) results. CSE reverses the effect of the expansion of complex polynomials. Our approach taken in this paper reduces the overall number of multiplication and addition operations by 41.0% and 12.9% respectively in
<table>
<thead>
<tr>
<th>Implementation</th>
<th>DSP (%)</th>
<th>ALM (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>35</td>
<td>1015</td>
</tr>
<tr>
<td>Buffered [LSK11]</td>
<td>31(-11.4%)</td>
<td>960(-5.4%)</td>
</tr>
<tr>
<td>Poly. Opt.</td>
<td>22(-37.1%)</td>
<td>274(-73.0%)</td>
</tr>
<tr>
<td>Poly. + Arith. Opt.</td>
<td>0</td>
<td>4964(389.1%)</td>
</tr>
</tbody>
</table>

Table 5.3: FPGA Implementation Costs

comparison with the Direct Evaluation. For analytically analysis of the critical delay refer to Section 4.3. We used the data-flow graphs of each approach to generate the bit-widths required for each operation. We then synthesized these three approaches for Altera Cyclone V 5CEFA7F31C7 FPGA using Quartus 13 and simulated using Multisim. We also used our generator to apply the arithmetic optimization and synthesized under the same conditions. Table 5.3 lists out the number of DSP’s and adaptive logic modules (ALM) required for each implementation. Our approach saves 37.1% in DSP’s and 73.0% in logic blocks comparing to the direct evaluation and it beats the cost reduction from the best previous attempt (Buffered)[LSK11]. Table 5.4 shows the maximum frequency of each implementation. Our approach provides enhancements in the speed by 55.9% comparing to the direct approach [LSK11] due to algebraic and implementation techniques we introduced in Chapters 3 and 4. According to [alt07] the digital pre-distorters have a bandwith of 15 to 20 MHz, which puts our solution ahead in terms of speed. None of the approaches studied shared any multiplier or adder blocks. Our approach can be enhanced more by using sequential logic which should result in an increase in the maximum frequency. Additionally, by reducing the required maximum frequency and sharing multipliers in different phases of evaluating the polynomial, one can reduce the cost further. To test our approach further, we generated additional 4 sets of polynomials by changing the parameters of our model. This was an attempt to validate the advantage of our approach over previous attempts to more than one set of inputs. Since predistorters are sensitive to the operating temperature, in a real implementation the designer can include
different predistorters and switch between the outputs based on the associating temperature range. By modifying the parameters we tried to emulate the changes of our nonlinearity due to the temperature. The changes to the parameters are arbitrary and they do not rely on an actual temperature sensitivity model. Table 5.5 lists the multiplication involved in the different sets of polynomials generated with the respective parameters. The first row reflects the polynomials associated with the original parameters mentioned above. Our approach outperforms previous attempts in decreasing the number of multiplications which require the most area.
CHAPTER 6

Summary

We have presented optimization of polynomials using algebraic manipulations at the high level and digital arithmetic techniques at the implementation level. We have treated the polynomial optimization problem in abstract algebra and we have focused on Volterra series as our polynomial class which can be used to model non-linear systems. We have used techniques such as variables and common coefficient factorization, common subexpression elimination, and delayed products to optimize polynomials at the algebraic level. These have resulted in reduction of 41.0% and 12.9% in number of multiplication and addition respectively in a predistorter we have implemented. These reductions are in comparison with common subexpression elimination technique used in compilers. At implementation level, we have used redundant arithmetic to increase the speed of the polynomial evaluation. This has resulted in an increase of 19.7% in maximum frequency in comparison with the optimized design using conventional arithmetic and mapped to DSP blocks available in FPGA. To avoid manual generation of such systems, we have developed a generator. This generator parses the polynomials into expression trees and automatically outputs Verilog HDL code. Using techniques such as redundant arithmetic, recoding, redundant truncation, and arrival time consideration has resulted in 72.7% increase in the speed of synthesized circuit in comparison with the optimized polynomial using our algebraic technique and synthesized using conventional arithmetic and avoiding DSP blocks. Finally we used our methods to develop a digital pre-distorter to reverse the effect of non-linearity
in a satellite communication system. Our optimization techniques have resulted in an increase of 55.9% in speed comparing to direct evaluation of Volterra series. The optimization techniques presented can be integrated into a High-Level Synthesis (HLS) environment to extract polynomials automatically. That would reduce the time designers have to spend optimizing polynomial datapaths.

6.1 Future Work

We present following items as future work to further optimize polynomials as well as any application that can be implemented using our techniques:

- The technique used for multiple-constant multiplication in [EL03, PSC96] which exposes common subexpressions among the constant multipliers can potentially reduce the size.

- Even though we have not studied power in this work, the reduction in number of operations due to our algebraic methods indirectly results in reduction of power. Further study would push the power consumption lower by considering the power cost function and slower implementations when feasible.

- We have used redundant representation throughout our implementation. As explained, some of the operands in multi-operand addition arrive early, and they can be good candidates to be implemented using conventional arithmetic. This would reduce the size as long as the conventional arithmetic does not become part of the critical delay.

- Even though our generator was built to output HDL, it can be modified to map the polynomials to other architectures such as data-flow architecture or software.

- We did not put any limit on the polynomials we used (Volterra), so any application or system that can be modeled using Volterra series would benefit
from the work presented.
APPENDIX A

Appendix

A.1 Abstract Algebra

Abstract algebra is the area of mathematics that studies algebraic structures, such as groups, rings, fields, and etc.

Definition 11. An algebraic structure comprises one or more sets, called underlying sets, closed under one or more operations, satisfying some defined axioms [Wik11].

For the purpose of this research, we briefly include the definition of algebraic structures referenced and utilized throughout this document.

A.1.1 Basic Algebraic Structures

Definition 12. A monoid is a set, $S$, together with a binary operation "," which combines any two elements $a$ and $b$ from $G$ to form another element, denoted as $a \cdot b$. To qualify as a monoid, the set and operation, $(S, \cdot)$, must fulfill three axioms [Her75]:

- **Closure**
  
  $a \cdot b \in S \quad \forall a, b \in S$

  or $(\cdot : S \times S \rightarrow S)$

- **Associativity**

  $(a \cdot b) \cdot c = a \cdot (b \cdot c) \quad \forall a, b, c \in S$
• **Identity element**

\[ \exists! e \in S : e \cdot a = a \cdot e = a \quad \forall a \in S \]

The identity element of a monoid is denoted as \(1_S\) or simply \(1\).

**Definition 13.** A group is a set, \(G\), together with a binary operation \(\cdot\). To qualify as a group, the set and the operation, \((G, \cdot)\) must be a Monoid (Closed, Associative, and it has Identity element) and must fulfill a fourth axiom [Her75]:

• **Inverse element**

\[ \exists b \in G : a \cdot b = b \cdot a = 1_G \quad \forall a \in G \]

The inverse of element \(a\) when the group is represented as \(+\) \((G, +)\) is usually represented as \(-a\).

The order in which the group operation is performed is important and in the general case \(a \cdot b \neq b \cdot a\). An Abelian group, is defined as a group which satisfies a fifth axiom:

• **Commutativity**

\[ a \cdot b = b \cdot a \quad \forall a, b \in G \]

**Definition 14.** A ring is a set \(R\) together with two binary operations \(+\) and \(\cdot\) called addition and multiplication. To qualify as a ring, the set and two operations, \((R, +, \cdot)\), must satisfy the following axioms (Ring Axioms):

• \((R, +)\) (called additive group of the ring) is an Abelian group (Closed, Associative, Commutative, it has Identity element and Inverse elements)

• \((R, \cdot)\) is a Monoid (Closed, Associative, and it has Identity element)

• **Distributivity**

\[ a \cdot (b + c) = (a \cdot b) + (a \cdot c) \quad \forall a, b, c \in G \]

\[ (a + b) \cdot c = (a \cdot c) + (b \cdot c) \quad \forall a, b, c \in G \]
A ring which also satisfies **Commutativity** under multiplication (·) is called a **Commutative Ring**.

**Definition 15.** A subring of a ring \( R \) is a subset \( S \) of it that is also a ring when the two binary operations + and · are restricted to this subset.

**Definition 16.** A ring extension of the ring \( S \) is a ring \( R \) where \( S \) is a subring of \( R \). Ring extension of \( S \) is written as \( \frac{R}{S} \).

### A.1.2 Polynomial Rings

**Definition 17.** The set of all polynomials with coefficients in the field \( F \), represented as \( F[X] \) forms a commutative ring and is called the ring of polynomials over \( F \) [Irv04]. The symbol \( X \) represents a set of variables \( X = (x_1, x_2, \ldots, x_k) \) (the polynomial ring is denoted as \( F[x_1, x_2, \ldots, x_n] \)).

**Definition 18.** In a give polynomial ring \( F[x_1, x_2, \ldots, x_k] \) monomial order is a total order (\(<\)) defined on the set of all monomials of the polynomial ring, where for any monomials \( a, b, \) and \( c \) we have:

- **Compatible with multiplication** \( a < b \implies ac < bc \)
- **Minimal element** \( 1 < u \)

**Definition 19.** A subset \( I \) of a ring \( R \) is an ideal if \( (I, +) \) is a subgroup (forms a group by itself) and the subset \( I \) follows following axioms [LCO07]:

- **Absorption of elements of \( R \) under multiplication**
  
  \[
  x \cdot r \in I \quad \forall x \in I, \forall r \in R \\
  r \cdot x \in I \quad \forall x \in I, \forall r \in R 
  \]

For example every finite set of polynomials \( P = \{p_1, p_2, \ldots, p_k\} \subset F[x_1, x_2, \ldots, x_n] \) generates an ideal:
\[ < P > = \left\{ \sum_{i=1}^{k} a_i \cdot p_i \mid a_i \in F[x_1, x_2, \ldots, x_n] \right\} \]

The generating set \( P \) is called the basis for this ideal. It can be easily shown that this definition is an ideal:

- \((< P >, +)\) is a group:
  
  \[ 0 \in < P > \text{ by selecting all } a_i = 0 \text{ in the definition} \]

  \[ \begin{align*}
  r + s &= \sum_{i=1}^{k} r_i \cdot p_i + \sum_{i=1}^{k} s_i \cdot p_i = \sum_{i=1}^{k} (r_i + s_i) \cdot p_i \in < P > & \forall r, s \in F[x_1, x_2, \ldots, x_n]
  \end{align*} \]

- Absorption of elements of \( R \) under multiplication:
  
  \[ \begin{align*}
  r \cdot s &= \left( \sum_{i=1}^{k} r_i \cdot p_i \right) \cdot s = \sum_{i=1}^{k} (r_i \cdot s) \cdot p_i \in < P > & \forall r \in < P >, \forall s \in F[x_1, x_2, \ldots, x_n]
  \end{align*} \] (Similarly for \( s \cdot r \))

### A.1.2.1 Ideal Membership Problem

Given a polynomial \( p \in F[x_1, x_2, \ldots, x_n] \) and an Ideal \(< Q >\) with basis \( Q = \{q_1, q_2, \ldots, q_k\} \subset F[x_1, x_2, \ldots, x_n]\), we would like to determine whether \( p \in < Q > \) or :

\[ p \in < Q > \iff \exists a_1, a_2, \ldots, a_k \in F[x_1, x_2, \ldots, x_n] : p = \sum_{i=1}^{k} a_i \cdot q_i \]

**Definition 20.** The product of the form \( \prod_{i=1}^{k} x_i^{a_i} \) where \( x_1, x_2, \ldots, x_k \) are the variables of the polynomial ring \( F[x_1, x_2, \ldots, x_n] \) and \( a_1, a_2, \ldots, a_k \) are non-negative integers, is called a monomial

**Definition 21.** Leading monomial of the polynomial \( p \in F[x_1, x_2, \ldots, x_n] \) with respect to a monomial ordering, such as lexicographical, is the monomial in \( p \) where its term is maximal under the monomial ordering. We denote this as \( \text{LM}(p) \) and its coefficient as \( \text{LC}(p) \). We define Leading term as \( \text{LT}(p) = \text{LC}(p) \cdot \text{LM}(p) \).
For example under lexicographical ordering ($>_{lex}$):

\[ p = 4x_1^3 + 6x_1x_2^4 + x_3 \]

\[ \text{LT}(p) = 4x_1^3 \quad \text{LC}(p) = 4 \quad \text{LM}(p) = x_1^3 \]

**Theorem A.1.1.** (Division Algorithm in $F[x_1, x_2, \ldots, x_n]$) Let $p$ a polynomial from the ring $F[x_1, x_2, \ldots, x_n]$ and $Q$ an ordered set $Q = (q_1, q_2, \ldots, q_k)$. Then:

\[ p = \sum_{i=1}^{k} a_i q_i + r \]

where $a_i, r \in F[x_1, x_2, \ldots, x_n]$ and either $r = 0$ or is a linear combination, with coefficients in $F$, of monomials, none of which is divisible by any of $\text{LT}(q_i)$. We call $r$ the remainder of $p$ on division by $Q$.

**Proof.** Refer to [LCO07]

This is a generalization of univariate polynomial division to multivariate polynomial division using a specific monomial ordering.

**Definition 22.** We define the operator $p \xrightarrow{Q} r$ as the reduction of $p$ under $Q$ defined by A.1.1.

It can be shown [LCO07] that $r$ is not unique and it can depend on the order of $Q$. More importantly the ideal membership problem using this definition can be represented as:

\[ p \xrightarrow{Q} = 0 \Rightarrow p \in < Q > \]

However, the converse is not always true for ideals. In the next section we will introduce a special class of ideals defined by basis called Gröbner basis where:

\[ p \in < \text{gröbner-basis}(Q) > \Rightarrow p \xrightarrow{\text{gröbner-basis}(Q)} = 0 \Rightarrow p \in < Q > \]
### A.1.2.2 Gröbner basis

**Definition 23.** An ideal basis $G \subset F[x_1, x_2, \ldots, x_n]$ is a Gröbner basis (with respect to a specified monomial ordering) if:

$$p \xrightarrow{G} 0 \iff p \in \langle G \rangle$$

**Definition 24.** A reduced Gröbner basis for a polynomial ideal $I$ is a Gröbner basis $G$ for $I$ such that:

- $\text{LC}(p) = 1 \ \forall p \in G$.
- For all $p \in G$, no monomial of $p$ lies in $\langle \text{LT}(G - \{p\}) \rangle$.

**Definition 25.** We define $S$-Polynomial of two polynomials $p$ and $q$ as:

$$S\text{-Poly}(p, q) = \text{LCM}(\text{LT}(p), \text{LT}(q)) \cdot \left( \frac{p_{\text{LT}(p)}}{\text{LT}(p)} - \frac{q_{\text{LT}(q)}}{\text{LT}(q)} \right)$$

Where $\text{LCM}$ is the least common multiple operator. $S\text{-Poly}$ by definition eliminates the leading terms from both $p$ and $q$.

**Theorem A.1.2.** (Buchberger’s Criterion) Let $I$ be a polynomial ideal basis. Then a basis $G = \{g_1, g_2, \ldots, g_t\} \subset F[x_1, x_2, \ldots, x_n]$ for $I$ is a Gröbner basis for $I$ if and if $S\text{-Poly}(g_i, g_j) \xrightarrow{G} 0 \ \forall i \neq j$

**Proof.** Refer to [LCO07]  

Theorem A.1.2 also hints an algorithm to calculate Gröbner basis, which is called *Buchberger’s Algorithm* and it’s presented in Algorithm A.1.1
Algorithm A.1.1 gröbner-basis(Q) [Buc06]

\[ G \leftarrow Q \]

\[ B \leftarrow G \times G \]

\textbf{while} \( B \neq \emptyset \) \textbf{do}

\((p, q) \leftarrow \text{a pair from } B\)

\[ B \leftarrow B \setminus \{(p, q)\} \]

\[ h \leftarrow \text{S-Poly}(p, q) \]

\textbf{if} \( h \neq 0 \) \textbf{then}

\[ B \leftarrow B \cup (G \times \{h\}) \]

\[ G \leftarrow G \cup \{h\} \]

\textbf{end if}

\textbf{end while}

\textbf{return} \( G \)
A.2 Functions and Operators

<table>
<thead>
<tr>
<th>Function</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>quo</td>
<td>$f, d$</td>
<td>Returns the quotient of division of $f$ over $d$</td>
</tr>
<tr>
<td>rem</td>
<td>$f, d$</td>
<td>Returns the remainder of division of $f$ over $d$</td>
</tr>
<tr>
<td>compatible</td>
<td>$p, c_1, c_2$</td>
<td>Returns true if decomposition on $c_1$ and then $c_2$ are allowed</td>
</tr>
<tr>
<td>add</td>
<td>$a, b$</td>
<td>Returns $a + b$</td>
</tr>
<tr>
<td>mult</td>
<td>$a, b$</td>
<td>Returns $a \times y$</td>
</tr>
<tr>
<td>gcd</td>
<td>$a, b$</td>
<td>Returns the greatest common divisor of $a$ and $b$</td>
</tr>
<tr>
<td>push</td>
<td>$L, a$</td>
<td>Push $a$ to the end of the list $L$</td>
</tr>
<tr>
<td>shift</td>
<td>$L$</td>
<td>Removes and returns the first element in list $L$</td>
</tr>
</tbody>
</table>

Table A.1: List of functions used
A.3 Verilog Blocks

Below we present the basic blocks we have implemented and used in our generator:

Full-Adder

```verilog
module FA (  
    input a,  
    input b,  
    input cin,  
    output sum,  
    output cout );  
assign sum = a ^ b ^ cin;  
assign cout = ( a & b) | ( a & cin ) | (b & cin);  
endmodule
```

[3 : 2] Adder (CSA)

```verilog
module ThreeTwo (  
    input [WIDTH−1:0] a,  
    input [WIDTH−1:0] b,  
    input [WIDTH−1:0] c,  
    output [WIDTH−1:0] sum,  
    output [WIDTH−1:0] carry // Total = sum + carry  
);  
parameter WIDTH = 4;  
genvar i;  
genenerate  
    for (i=0;i<WIDTH−1;i=i+1) begin: fa  
        FA fa ( .a( a[i] ) , .b( b[i] ) , .cin( c[i] ) , .sum( sum[i] ) , .cout ( carry[i+1] ) );  
    end  
endgenerate  
wire cout;  
FA fa last ( .a( a[WIDTH−1] ) , .b( b[WIDTH−1] ) , .cin( c[WIDTH−1] ) , .sum( sum[  
    WIDTH−1] ) , .cout( cout ) );  
assign carry[0]=0;  
endmodule
```
[3:2] Adder (CSA) with Truncation

```verilog
module ThreeTwoTruncate (  
    input [WIDTH:0] a,  
    input [WIDTH:0] b,  
    input [WIDTH:0] c,  
    output [WIDTH−1:0] sum,  
    output [WIDTH−1:0] carry  // Total = sum + carry  
);  
parameter WIDTH = 4;  

genvar i;  
generate  
for (i = 0; i < WIDTH−2; i = i + 1) begin: fa  
    FA fa (. a ( a [i] ) , . b( b [i] ) , . cin ( c [i] ) , . sum( sum [i] ) , . cout( carry [i+1] ) );  
end  
endgenerate  
wire c0;  
FA fa_before_last (. a ( a [WIDTH−2] ) , . b( b[WIDTH−2] ) , . cin ( c[WIDTH−2] ) , . sum( sum[WIDTH−2] ) , . cout( c0 ) );  
wire c1;  
wire s0;  
FA fa_last (. a ( a[WIDTH−1] ) , . b( b[WIDTH−1] ) , . cin ( c[WIDTH−1] ) , . sum( s0 ) , . cout( c1 ) );  
wire s1;  
wire cout;  
FA fa_after_last (. a( a[WIDTH] ) , . b( b[WIDTH] ) , . cin( c[WIDTH] ) , . sum( s1 ) , . cout( cout ) );  
assign carry [0] = 0;  
assign sig = c1 ^ s1;  
assign sum [WIDTH−1] = sig ^ s0;  
assign carry [WIDTH−1] = sig ^ c0;  
endmodule
```

[4 : 2] Compressor

```verilog
// 4:2 Compressor, Ercegovac Book. Page 145
```
module Compressor (
    input a,
    input b,
    input c,
    input d,
    input cin,
    output sum,
    output carry,
    output cout);
wire sel1, sel2;
assign sel1 = c ^ d;
Mux mux1 (.i0(d), .i1(b), .sel(sel1), .out(cout));
assign sel2 = sel1 ^ a ^ b;
Mux mux2 (.i0(a), .i1(cin), .sel(sel2), .out(carry));
assign sum = cin ^ sel2;
endmodule

[4 : 2] Adder (CSA)

module FourTwo (  
    input [WIDTH-1:0] a,
    input [WIDTH-1:0] b,
    input [WIDTH-1:0] c,
    input [WIDTH-1:0] d,
    output [WIDTH-1:0] sum,  // Total = sum + carry = a + b + c + d
    output [WIDTH-1:0] carry
);
parameter WIDTH = 4;
// Internal Carry
wire int_carry [WIDTH:0];
assign int_carry [0] = 0;

genvar i;
generate
for (i=0;i<WIDTH-1;i=i+1) begin: comp
    Compressor comp( .a( a[i] ), .b( b[i] ), .c( c[i] ), .d( d[i] ), .cin( int_carry[i] ), .sum( sum[i] ), .carry( carry[i+1] ), .cout( int_carry[i+1] ) );
end
generate
wire cout;
wire c1, s1;
Compressor comp_last( .a( a[WIDTH-1] ), .b( b[WIDTH-1] ), .c( c[WIDTH-1] ), .d( d[WIDTH-1] ), .cin( int_carry[WIDTH-1] ), .sum( sum[WIDTH-1] ), .carry( c1 ),
    .cout( s1 ) );
assign carry[0]=0;
endmodule

[3 : 2] Adder (CSA) with Truncation

module FourTwoFixed ( 
    input [WIDTH:0] a,
    input [WIDTH:0] b,
    input [WIDTH:0] c,
    input [WIDTH:0] d,
    output [WIDTH-1:0] sum,
    output [WIDTH-1:0] carry
    // Total = sum + carry = a + b + c + d
); 
parameter WIDTH = 4;

    // Internal Carry
wire int_carry[WIDTH+1:0];
assign int_carry[0] = 0;

    genvar i;
generate
    for (i=0;i<WIDTH-2;i=i+1) begin: comp
        Compressor comp( .a( a[i] ), .b( b[i] ), .c( c[i] ), .d( d[i] ),
            cin( int_carry[i] ), .sum( sum[i] ), .carry( carry[i+1] ),
            cout( int_carry[i+1] ) );
    end
endgenerate

wire c0;
Compressor comp_before_last( .a( a[WIDTH-2] ), .b( b[WIDTH-2] ), .c( c[WIDTH-2] ),
    .d( d[WIDTH-2] ), .cin( int_carry[WIDTH-2] ), .sum( sum[WIDTH-2] ), .carry( c0 ),
    .cout( int_carry[WIDTH-1] ) );

wire c1;
wire s0;
Compressor comp_last (a[WIDTH−1], b[WIDTH−1], c[WIDTH−1], d[WIDTH−1], i[n] carry[WIDTH−1], sum s0, carry c1, cout(int carry[WIDTH]));

wire s1;
wire cout;
Compressor comp_after_last (a[WIDTH], b[WIDTH], c[WIDTH], d[WIDTH], i[n] carry[WIDTH], sum s1, carry cout, cout(int carry[WIDTH+1]));
assign carry[0]=0;
wire sig = c1 ^ s1;
assign sum[WIDTH−1] = s0 ^ sig;
assign carry[WIDTH−1] = c0 ^ sig;
endmodule

Inverter with Enable Signal

module NotE (input [WIDTH−1:0] inp, input inv, output [WIDTH−1:0] out);
parameter WIDTH = 1;

genvar i;
generate
  for (i=0;i<WIDTH;i=i+1) begin: n
    assign out[i] = inv ^ inp[i];
  end
endgenerate
endmodule

2’s Complement to Minimally Redundant Radix-4 Recoder

// 2’s complement to Minimally redundant Radix-4 Recoder
module TC_Recoder (input [WIDTH−1:0] m, output [WIDTH/2−1:0] sgn, output [WIDTH/2−1:0] p1, output [WIDTH/2−1:0] p2);
parameter WIDTH = 8;
assign t[0] = 0;

TC_RecoderBlock rb_first(.y2i1(m[1]), .y2i(m[0]), .y2i_1(0), .sgn(sgn[0]), .p1(p1[0]), .p2(p2[0]));
genvar i;
generate
   for (i=1;i<WIDTH/2;i=i+1) begin: rb
      TC_RecoderBlock rb(.y2i1(m[i*2+1]), .y2i(m[i*2]), .y2i_1(m[i*2-1]), .sgn(sgn[i]), .p1(p1[i]), .p2(p2[i]));
   end
gendgenerate
endmodule

module TC_RecoderBlock(
   input y2i1, input y2i, input y2i_1, output sgn, output p1, output p2);
assign sgn = y2i1;
assign p1 = y2i ^ y2i_1;
assign p2 = y2i1 & ~y2i & ~y2i_1 & ~y2i1 & y2i & y2i_1;
endmodule

Carry-Save to Minimally Redundant Radix-4 Recoder

// CS to Minimally redundant Radix 4 Recoder
module CS_Recoder(
   input [WIDTH-1:0] c, input [WIDTH-1:0] s, output [WIDTH/2:0] sgn, output [WIDTH/2:0] p1, output [WIDTH/2:0] p2);
parameter WIDTH = 8;

wire [WIDTH/2:0] tmp_p_2;
wire [WIDTH/2:0] tmp_p_1;
wire [WIDTH/2:0] tmp_p1;
wire [WIDTH/2:0] tmp_p2;
wire h[WIDTH/2:0];
assign h[0] = 0;
wire t[WIDTH/2:0];
assign t[0] = 0;
genvar i;
generate
    for (i=0;i<WIDTH/2−1;i=i+1) begin: rb
        CS_RecorderBlock rb ( .c1 ( c[i*2+1] ), .c0( c[i*2] ), .s1( s[i*2+1] ) , .s0( s[i*2] ) , .h( h[i] ) , .t ( t[i] ) , .h1( h[i+1] ) , .t1( t[i+1] ) , .p2( tmp_p2[i] ) , .p1( tmp_p1[i] ) , .p1( tmp_p1[i] ) );
    end
endgenerate
CS_RecorderSignBlock rb_last ( .c1( c[WIDTH−1] ) , .c0( c[WIDTH−2] ) , .s1( s[WIDTH−1] ) , .s0( s[WIDTH−2] ) , .h( h[WIDTH/2−1] ) , .t ( t[WIDTH/2−1] ) , .h1( h[WIDTH/2] ) , .t1( t[WIDTH/2] ) , .p2( tmp_p2[WIDTH/2−1] ) , .p1( tmp_p1[WIDTH/2−1] ) , .p1( tmp_p1[WIDTH/2−1] ) );
assign tmp_p2[WIDTH/2] = 0;
assign tmp_p1[WIDTH/2] = h[WIDTH/2] & ^t[WIDTH/2];
assign tmp_p1[WIDTH/2] = ^h[WIDTH/2] & t[WIDTH/2];
assign tmp_p2[WIDTH/2] = 0;
generate
    for (i=0;i<=WIDTH/2;i=i+1) begin: out
        assign sgn[i] = tmp_p1[i] | tmp_p2[i];
        assign p1[i] = tmp_p1[i] | tmp_p1[i];
        assign p2[i] = tmp_p2[i] | tmp_p2[i];
    end
endgenerate
endmodule

module CS_RecorderBlock ( 
    input c1,
    input c0,
    input s1,
    input s0,
    input h,
    input t,
    output h1,
    output t1,
    output p2,
    output p1,
module CS_RecoerSignBlock (  
    input c1,  
    input c0,  
    input s1,  
    input s0,  
    input h,  
    input t,  
    output h1,  
    output t1,  
    output p2,  
    output p1,  
    output p2);  
assign h1 = c1 & s1;  
// z = 2z_1 + z_0  
assign z11 = c1 `^` s1;  
// z_1^1  
assign z10 = ~(c1 `^` s1);  
// z_1^0  
assign z02 = c0 & s0;  
// z_0^2  
assign z01 = c0 `^` s0;  
// z_0^1  
assign z00 = ~c0 & ~s0;  
// z_0^0  
assign t1 = (z01 & h) | z02 | z11;  
assign w_2 = (z00 & z11 & ~h) | (z02 & z10 & ~h) | (z01 & z10 & h);  
assign w_1 = (z00 & z11 & h) | (z02 & z10 & h) | (z01 & z11 & ~h);  
assign w0 = (z02 & z11 & ~h) | (z01 & z11 & h) | (z00 & z10 & ~h);  
assign w1 = (z00 & z10 & h) | (z01 & z10 & ~h) | (z02 & z11 & h);  
assign p_2 = w_2 & ~t;  
assign p_1 = (w_2 & t) | (w_1 & ~t);  
assign p1 = (w0 & t) | (w1 & ~t);  
assign p2 = w1 & t;  
endmodule
assign \( t_1 = (z_{01} \& h) \mid z_{02} \mid z_{11}; \)

assign \( w_2 = (z_{00} \& z_{11} \& \neg h) \mid (z_{02} \& z_{10} \& \neg h) \mid (z_{01} \& z_{10} \& h); \)
assign \( w_1 = (z_{00} \& z_{11} \& h) \mid (z_{02} \& z_{10} \& h) \mid (z_{01} \& z_{11} \& \neg h); \)
assign \( w_0 = (z_{02} \& z_{11} \& \neg h) \mid (z_{01} \& z_{11} \& h) \mid (z_{00} \& z_{10} \& \neg h); \)
assign \( w_1 = (z_{00} \& z_{10} \& h) \mid (z_{01} \& z_{10} \& \neg h) \mid (z_{02} \& z_{11} \& h); \)

assign \( p_2 = w_2 \& \neg t; \)
assign \( p_1 = (w_2 \& t) \mid (w_1 \& \neg t); \)
assign \( p_1 = (w_0 \& t) \mid (w_1 \& \neg t); \)
assign \( p_2 = w_1 \& t; \)

endmodule
REFERENCES


[Qua] “Quartus II Web Edition Software.”


