Exploiting redundancy in modulus replication inner product processors.

Marjan Shahkarami

University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation

https://scholar.uwindsor.ca/etd/2035

This online database contains the full-text of PhD dissertations and Masters’ theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®
Exploiting Redundancy in Modulus Replication Inner Product Processors

by

Marjan Shahkarami

A Dissertation
Submitted to the College of Graduate Studies and Research through the Department of Electrical and Computer Engineering in partial fulfillment of the requirements for the Degree of Doctor of Philosophy at the University of Windsor

Windsor, Ontario, Canada
1999
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.
© 1999 Marjan Shahkarami

All Rights Reserved. No part of this document may be reproduced, stored or otherwise retained in a retrieval system or transmitted in any form, on any medium or by any means without the prior written permission of the author.
To my parents for their unending support, and to Mike for understanding.
List of Symbols

\[ \in \quad a \in S \quad \text{membership} \]
\[ \emptyset \quad \text{empty set} \]
\[ \cdot \quad \text{binary operation} \]
\[ S \quad \text{set} \]
\[ <R, +, \cdot> \quad \text{Ring} \]
\[ e \quad \text{identity element for groups and rings} \]
\[ a^{-1}, -a \quad \text{inverse of a (element) for the operation under consideration} \]
\[ \mathbb{Z} \quad \text{set of integers} \]
\[ \langle G, \ast \rangle \quad \text{Group} \]
\[ \mathbb{Z}_n \quad \text{cyclic group \{0, 1, ..., n-1\} under addition modulo } n \]
\[ C_n \quad \text{Cyclic Group of order } n \]
\[ |S| \quad \text{order of } S \]
\[ p \quad \text{prime number} \]
\[ \oplus_m \quad \text{addition mod } m \]
\[ \otimes_m \quad \text{multiplication mod } m \]
\[ X \quad \text{indeterminate} \]
\[ A(X) \quad \text{polynomial in } X \]
\[ \deg f(x), d \quad \text{degree of a polynomial in } x \]
\[ R[X] \quad \text{ring of polynomials in } X \text{ over } R \]
\[ \gcd \quad \text{greatest common divisor} \]
\[ a \equiv b \mod m \quad \text{congruence of } a \text{ and } b \]
$|a|_m$  \hspace{1cm} \textit{residue of a modulo m}

$R(M)$ \hspace{1cm} \textit{finite ring modulo M}

g \hspace{1cm} \textit{generator}

$GF(p^n)$ \hspace{1cm} \textit{Galois field of order} $p^n$

$\varphi:A \rightarrow B$ \hspace{1cm} \textit{mapping from A to B}

$\varphi(a)$ \hspace{1cm} \textit{map of a}

$<F,+,.\cdot,0,1>$ \hspace{1cm} \textit{Field}

$\text{char}(F)$ \hspace{1cm} \textit{characteristic of F}

$R_1 \times R_2$ \hspace{1cm} \textit{cross-product ring}

$D(A)$ \hspace{1cm} \textit{Diminished - l representation of A}$
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Arithmetic-Logic unit</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complementary Metal Oxide Semiconductor</td>
</tr>
<tr>
<td>CRNS</td>
<td>Complex Residue Number System</td>
</tr>
<tr>
<td>CRT</td>
<td>Chinese Remainder Theorem</td>
</tr>
<tr>
<td>CSA</td>
<td>Carry Save Adder</td>
</tr>
<tr>
<td>DI</td>
<td>Diminished-1</td>
</tr>
<tr>
<td>DFT</td>
<td>Discrete Fourier transform</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>EMODL</td>
<td>Enhanced Multiple Output Domino Logic</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Arrays</td>
</tr>
<tr>
<td>FRNS</td>
<td>Flexible Residue Number System</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit</td>
</tr>
<tr>
<td>LSB</td>
<td>Least Significant Bit</td>
</tr>
<tr>
<td>MAC</td>
<td>Multiply-Accumulate</td>
</tr>
<tr>
<td>MMRNS</td>
<td>Modulus Replication Number System</td>
</tr>
<tr>
<td>MOS</td>
<td>Metal-Oxide Semiconductor</td>
</tr>
<tr>
<td>MQRNS</td>
<td>Modified Quadratic Residue Number System</td>
</tr>
<tr>
<td>MS</td>
<td>Most Significant</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bit</td>
</tr>
<tr>
<td>NAN</td>
<td>Not a Number</td>
</tr>
<tr>
<td>PE</td>
<td>Processing Element</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>PRNS</td>
<td>Polynomial Residue Number System</td>
</tr>
<tr>
<td>QRNS</td>
<td>Quadratic Residue Number System</td>
</tr>
<tr>
<td>QLRNS</td>
<td>Quadratic-Like Residue Number System</td>
</tr>
<tr>
<td>RNS</td>
<td>Residue Number System</td>
</tr>
<tr>
<td>ROM</td>
<td>Read only Memory</td>
</tr>
<tr>
<td>SIA</td>
<td>Semiconductor Industry Association</td>
</tr>
<tr>
<td>TSPC</td>
<td>True Single Phase Clock</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large Scale Integration</td>
</tr>
<tr>
<td>WSI</td>
<td>Wafer Scale Integration</td>
</tr>
</tbody>
</table>
There are several people who deserve my sincere thanks for their generous contributions to this dissertation.

I would first like to express my sincere gratitude and appreciation to Dr. G. A. Jullien, my supervisor for his invaluable guidance and constant support throughout the course of this thesis work. I am grateful to Genum Corp. for providing funding and support on this project. I would also like to thank my committee members Dr. M.A. Hasan, Dr. W.C. Miller, Dr. Arunita Jaekel, and Dr. Majid Ahmadi.

I would also like to recognize the following individuals and corporations for their contributions: Roberto Muscedere for his time and comments on the design and implementation issues. CMC for providing and supporting the design software and computing hardware which made this project possible. Micronet R&D for providing financial and networking support.

Finally I would like to thank all my colleagues at the VLSI Research Group for all the happy memories during the time spent working on this thesis and my stay in Windsor.
Abstract

This thesis presents a new mapping strategy and modified architectures for implementing general purpose inner product computations, using enhanced Fermat ALU theory. The structure is based on a direct product finite polynomial ring mapping of a redundant binary representation of the input data; in effect we exploit the double redundancy of the input representation and the mapped polynomial representation. By exploiting this redundancy, with attendant reductions in coefficient growth due to polynomial multiplication, considerably reduction in the probability of overflow error is achieved.

The redundant property of the polynomial map is used to optimize the input data. By allowing a mix of positive and negative coefficients to represent any number, regardless of sign, we can reduce the maximum value of the coefficient by as much as half. This is sufficient to reduce the probability of overflow to acceptable levels using only single modulus computations, with considerable reduction in computational hardware.

This thesis demonstrates, for the case of FIR filter inner product applications, that this new approach allows the implementation of reasonable filter lengths using only a Mod 257 ALU. The probability of overflow in the finite field channels is considerably reduced compared to an implementation without the enhanced mapping. This results in less hardware and less power dissipation and, due to the additional binary channel, an increase in the output dynamic range.

In terms of the efficacy of this new technique, area and power costs for the 53-tap design have been estimated and a complete floorplan and HDL simulations are presented.
# Table of Contents

## Chapter 1: Introduction

1.1 Introduction ................................................................................................................. 1  
1.2 VLSI Implementation of Special Purpose DSPs ..................................................... 3  
1.2.1 Residue Number System .................................................................................... 5  
1.3 Modulus Replication Number System ................................................................. 7  
1.4 Thesis Objectives ....................................................................................................... 8  
1.5 Thesis Organization .................................................................................................... 8  

## Chapter 2: Number Theory for Digital Signal Processing

2.1 Introduction ................................................................................................................. 9  
2.2 Introductory Number Theory .................................................................................. 10  
2.2.1 Groups ............................................................................................................... 10  
2.2.2 Isomorphism and Homorphisms ...................................................................... 11  
2.2.3 Cyclic Groups .................................................................................................... 12  
2.2.4 Rings and Fields ............................................................................................... 13  
2.2.5 Direct Product Rings ....................................................................................... 16  
2.2.6 Polynomial Rings ............................................................................................. 17  
2.2.7 Ideals and Quotient Rings ................................................................................ 20  
2.3 Polynomial Based Mappings .................................................................................. 21  
2.3.1 Quadratic Residue Number System ............................................................... 22  
2.3.2 Polynomial Residue Number System ............................................................. 23  
2.3.3 Moduli Replication RNS .................................................................................. 26  
2.4 Polynomial mapping comparisons ......................................................................... 34  
2.5 Summary .................................................................................................................... 36  

## Chapter 3: Redundant Polynomial Mapping in MRRNS

3.1 Introduction ................................................................................................................. 38
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>Modulus Replication</td>
<td>39</td>
</tr>
<tr>
<td>3.3</td>
<td>Modular Overflow Error of Inner Products</td>
<td>41</td>
</tr>
<tr>
<td>3.4</td>
<td>Polynomial Representation</td>
<td>43</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Single Indeterminate</td>
<td>43</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Multiple Indeterminates</td>
<td>47</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Comparisons</td>
<td>51</td>
</tr>
<tr>
<td>3.5</td>
<td>Binary Representation</td>
<td>52</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Unsigned binary number</td>
<td>52</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Signed binary number</td>
<td>52</td>
</tr>
<tr>
<td>3.5.3</td>
<td>One’s complement</td>
<td>54</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Two’s complement</td>
<td>55</td>
</tr>
<tr>
<td>3.5.5</td>
<td>Signed digit representation [74]</td>
<td>56</td>
</tr>
<tr>
<td>3.5.6</td>
<td>Comparison</td>
<td>60</td>
</tr>
<tr>
<td>3.6</td>
<td>Enhanced Polynomial Representation</td>
<td>61</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Overflow Error Analysis</td>
<td>68</td>
</tr>
<tr>
<td>3.7</td>
<td>Summary</td>
<td>75</td>
</tr>
</tbody>
</table>

Chapter 4  
Architecture  

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>76</td>
</tr>
<tr>
<td>4.2</td>
<td>Index Calculus Residue System</td>
<td>77</td>
</tr>
<tr>
<td>4.3</td>
<td>Diminished -1 Addition</td>
<td>79</td>
</tr>
<tr>
<td>4.4</td>
<td>Original Fermat ALU</td>
<td>82</td>
</tr>
<tr>
<td>4.5</td>
<td>A New Half index Domain MAC</td>
<td>84</td>
</tr>
<tr>
<td>4.5.1</td>
<td>ROM size reduction</td>
<td>84</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Modification for Diminished-1 Accumulation</td>
<td>86</td>
</tr>
<tr>
<td>4.6</td>
<td>Summary</td>
<td>87</td>
</tr>
</tbody>
</table>

Chapter 5  
A MRRNS FIR Array Case Study  

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>89</td>
</tr>
<tr>
<td>5.2</td>
<td>Input Mapping</td>
<td>89</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Polynomial mapper</td>
<td>89</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Evaluation map</td>
<td>91</td>
</tr>
<tr>
<td>5.3</td>
<td>Computational Channels</td>
<td>93</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Finite ring channels</td>
<td>93</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Binary Channel</td>
<td>95</td>
</tr>
<tr>
<td>5.4</td>
<td>Output mapper</td>
<td>96</td>
</tr>
<tr>
<td>5.5</td>
<td>Final Adder</td>
<td>97</td>
</tr>
<tr>
<td>5.6</td>
<td>FIR Array Floorplan</td>
<td>98</td>
</tr>
<tr>
<td>5.7</td>
<td>Example of a 53 TAP filter Design</td>
<td>99</td>
</tr>
<tr>
<td>5.7.1</td>
<td>Input Mapping Stage</td>
<td>100</td>
</tr>
<tr>
<td>5.7.2</td>
<td>Evaluation Map</td>
<td>104</td>
</tr>
<tr>
<td>5.7.3</td>
<td>Computational Channel</td>
<td>105</td>
</tr>
<tr>
<td>5.7.4</td>
<td>Output Stage</td>
<td>107</td>
</tr>
<tr>
<td>D.1</td>
<td>Introduction</td>
<td>315</td>
</tr>
<tr>
<td>D.2</td>
<td>Adder Design</td>
<td>315</td>
</tr>
<tr>
<td>D.3</td>
<td>ROM Design</td>
<td>318</td>
</tr>
<tr>
<td>D.4</td>
<td>Latch Design</td>
<td>321</td>
</tr>
<tr>
<td>D.5</td>
<td>Fermat ALU Layout</td>
<td>323</td>
</tr>
<tr>
<td>D.6</td>
<td>Comparison with a Binary MAC</td>
<td>325</td>
</tr>
</tbody>
</table>
List of Figures

Figure 2.1  Rings and Homorphisms of the MRRNS ...........................................27
Figure 2.2  Embedded RNS in MRRNS .................................................................30
Figure 3.1  Integer map from polynomial coefficients .........................................42
Figure 3.2  Plot of LHS of Eqn. (3.12), against X and B ......................................46
Figure 3.3  Plot of the number of inner products vs. X and B .................................47
Figure 3.4  Polynomial mapping for sign and magnitude representation ..................53
Figure 3.5  Polynomial mapping for one's complement representation ....................56
Figure 3.6  Polynomial mapping for signed digit (case 1) ......................................58
Figure 3.7  Polynomial mapping for signed digit (case 2) ......................................59
Figure 3.8  Original map ....................................................................................66
Figure 3.9  Map 1 ..............................................................................................66
Figure 3.10 Map 2 ..............................................................................................66
Figure 3.11 Map 3 ..............................................................................................67
Figure 3.12 Map 4 ..............................................................................................67
Figure 3.13 Map 5 ..............................................................................................67
Figure 3.14 Map 6 ..............................................................................................68
Figure 3.15 Map 7 ..............................................................................................68
Figure 3.16 Uniform Distribution of Input Data ....................................................69
Figure 3.17 Histogram of filter coefficients ..........................................................71
Figure 3.18 Histogram of polynomial coefficients without enhanced mapping ..........72
Figure 3.19 Histogram of polynomial coefficients with enhanced mapping .............73
Figure 4.1  Modulo(257)-MAC Implementation in the half-index domain ...............84
Figure 4.2  New MAC block diagram ..................................................................87
Figure 5.1  Block diagram of the input polynomial mapper .....................................90
Figure 5.2  Evaluation map ................................................................................93
Figure 5.3  Block Diagram of Fermat ALU ..............................................................94
Figure 5.4  Minimized ROM .............................................................................94
Figure 5.5  Block diagram of output mapper ........................................................97
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6</td>
<td>CSA array for the final polynomial</td>
<td>98</td>
</tr>
<tr>
<td>5.7</td>
<td>Floorplan of Enhanced Fermat ALU Array</td>
<td>99</td>
</tr>
<tr>
<td>5.8</td>
<td>Floorplan of the Original Fermat ALU</td>
<td>99</td>
</tr>
<tr>
<td>5.9</td>
<td>Enhanced polynomial mapper</td>
<td>101</td>
</tr>
<tr>
<td>5.10</td>
<td>Switching tree ROM schematic</td>
<td>102</td>
</tr>
<tr>
<td>5.11</td>
<td>Switching tree ROM layout</td>
<td>102</td>
</tr>
<tr>
<td>5.12</td>
<td>Spice results from switching tree</td>
<td>103</td>
</tr>
<tr>
<td>5.13</td>
<td>TSPC latch SPICE results</td>
<td>103</td>
</tr>
<tr>
<td>5.14</td>
<td>Evaluation Map</td>
<td>105</td>
</tr>
<tr>
<td>5.15</td>
<td>Single adder stage</td>
<td>106</td>
</tr>
<tr>
<td>5.16</td>
<td>Modified Fermat ALU</td>
<td>107</td>
</tr>
<tr>
<td>5.17</td>
<td>Inverse polynomial map</td>
<td>108</td>
</tr>
<tr>
<td>5.18</td>
<td>Final Adder</td>
<td>109</td>
</tr>
<tr>
<td>5.19</td>
<td>53-tap FIR floorplan</td>
<td>109</td>
</tr>
<tr>
<td>C.1</td>
<td>Fermat ALU Verilog Simulation</td>
<td>301</td>
</tr>
<tr>
<td>C.2</td>
<td>Input Mapper Schematic</td>
<td>303</td>
</tr>
<tr>
<td>C.3</td>
<td>Input Mapper Verilog Simulation</td>
<td>304</td>
</tr>
<tr>
<td>C.4</td>
<td>Output Mapper Schematic</td>
<td>305</td>
</tr>
<tr>
<td>C.5</td>
<td>Output Mapper Layout</td>
<td>305</td>
</tr>
<tr>
<td>C.6</td>
<td>Pipeline Adder Schematic</td>
<td>307</td>
</tr>
<tr>
<td>C.7</td>
<td>Pipeline Adder Verilog Simulation</td>
<td>308</td>
</tr>
<tr>
<td>D.1</td>
<td>A single-bit level in the EMODL tree</td>
<td>316</td>
</tr>
<tr>
<td>D.2</td>
<td>A 4-bit EMODL adder tree</td>
<td>317</td>
</tr>
<tr>
<td>D.3</td>
<td>Accelerating X-connector</td>
<td>318</td>
</tr>
<tr>
<td>D.4</td>
<td>4(n) Adder Tree cascade with X-connectors</td>
<td>318</td>
</tr>
<tr>
<td>D.5</td>
<td>Dynamic sense amplifier</td>
<td>319</td>
</tr>
<tr>
<td>D.6</td>
<td>Dynamic decoder unit</td>
<td>320</td>
</tr>
<tr>
<td>D.7</td>
<td>ROM layout</td>
<td>321</td>
</tr>
<tr>
<td>D.8</td>
<td>A Buffered TSPC Latch</td>
<td>322</td>
</tr>
<tr>
<td>D.9</td>
<td>Domino stages with TSPC latch</td>
<td>323</td>
</tr>
<tr>
<td>D.10</td>
<td>Layout of Fermat 257</td>
<td>324</td>
</tr>
<tr>
<td>D.11</td>
<td>Low cycle rate test and Delay test</td>
<td>325</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Introduction

Digital signal processing (DSP), in the strict sense of the term, refers to the digital electronic processing of signals such as sound, radio, and microwaves [120]. DSPs encompass a broad spectrum of applications, such as digital filtering, image processing, signal compression, etc.

The first practical general purpose real-time DSP systems emerged in the late 1970s and used bipolar "bit-slice" components. Large quantities of these building-block chips were needed to design a system, at considerable effort and expense. Uses were limited to esoteric high-end technology, such as military and space systems. The economics began to change in the early 80s with the advent of single-chip MOS (Metal-Oxide Semiconductor) DSPs. Cheaper and easier to design-in than building blocks, these "monolithic" processors meant that digital signal processing could be cost-effectively integrated into an array of ordinary products. The design style was in similar direction to a single chip general purpose processor design, with multiple fast floating point Arithmetic logic units (ALUs) on board. The chips are easy to use, with the algorithms programmed into the chip. This ease of programmability
for a wide variety of algorithms was at the cost of sacrificing speed/power on silicon.

A digital filter is a digital system that can be used to filter discrete-time signals, in other words modify, manipulate, and reshape the frequency spectrum of the input signal. It can be implemented by means of software on a general purpose architecture, or by means of dedicated hardware such as special purpose DSPs. A bandlimited continuous signal can be converted to into a discrete signal by means of sampling. Conversely the discrete signal so generated can be used to regenerate the original continuous signal by means of interpolation, by virtue of Shannon's sampling theorem. As a consequence, hardware digital filters can be used to perform real-time filtering tasks, which only two decades ago were performed almost exclusively by analog filters. The design of special purpose DSPs for filtering applications is the focus of this thesis work.

Special purpose DSP chips also saw their birth in the late seventies. These special purpose architectures were programmable in that certain algorithmic parameters could be modified, such as with a finite impulse response (FIR) filter design with programmable coefficients. The difference between the special purpose DSP and the general purpose DSPs was that the specific algorithms could be performed much faster and with greater throughput with the special purpose DSPs but at the cost of fixed functionality.

In digital signal processing, a vast majority of the arithmetic functions required are of the inner product type (such as FIR filters, and transforms such as the Discrete Fourier Transform DFT), and thus the multiply-accumulate tends to be the central operation in many types of DSP systems. Thus special purpose DSP hardware implementations of these signal processing functions invariably results in attempts to optimize the multiply-accumulator pipeline. Pipelining is a well know architectural technique for increasing the throughput of a system implementing a computational task composed of several independent subtasks to be sequentially performed on each element of input data. These subtasks can range anywhere from a large set of operations to a single operation, or even the bit-sliced activities involved in a single operation.
Another way to classify DSP devices and applications is by their dynamic range. The
dynamic range is the spread of numbers, from small to large, that must be processed in the
course of an application. It takes a certain range of values, for instance, to describe the
entire waveform of a particular signal, from deepest valley to highest peak. The range may
get even wider as calculations are performed, generating larger and smaller numbers
through multiplication and division. The DSP device must have the capacity to handle the
numbers so generated. If it does not, the numbers may "overflow", destroying the results
of the computation. The processor's capacity is a function of its data width (i.e. the
number of bits it manipulates) and the type of arithmetic it performs (i.e., fixed or floating
point). Each type of processor is ideal for a particular range of applications.

1.2 VLSI Implementation of Special Purpose DSPs

With recent advances in VLSI technologies, very complex DSP algorithms can be cost
effectively implemented. But, at the same time, the design complexity to achieve high-
speed performance, area efficiency and reliability becomes a major challenge. The
fundamental limitation to implementing highly complex structures, is the cost of
communication relative to logic and storage. Communication is expensive in terms of chip
area. Most of the area of a chip is covered with wires on several levels with transistor
switches rarely taking more than five percent of the area on the lowest level. When it
comes to performance, communication is expensive in delay where non zero resistance of
wires, together with the parasitic distributed capacitance, imposes a delay in the wire
itself. This is becoming increasingly significant with smaller geometries. Communication
is also expensive in sending signals between chips, when package pin limitation, the area
used for bonding pads and pad drivers, is considered.

Finally, the dynamic power supplied to the chip, and dissipated in the circuit at the switch
capacitive signal nodes is typically dominated by the parasitic capacitance of the internal
wires, bonding pads and inter chip wires, rather than by the capacitance of the transistor
gates; thus both the cost and performance metrics of VLSI favour architectures in which
communication is localized. This principle of locality is seen at every level of VLSI
design. The semiconductor industry association (SIA) predicts that by 2010, industry will be manufacturing 800-million-transistor processors with thousands of pins, a 1,000-bit bus, and clock speeds over 2 GHz [18]. Such chips would produce a predicted maximum power of 180W; only doubling the power dissipation of current much less-dense chips. Major contributions in this drive for power reduction must come in the form of not only power management or judicious scrutiny of every milliwatt consumed but also a wide spectrum of technologies, ranging from fundamental device enhancement to enhanced computational algorithms, from different architectures to clever circuit techniques.

In VLSI, where memory and processing power are relatively cheap there is an emphasis on keeping the overall architecture as regular and modular as possible, thus reducing the overall complexity. If a structure can be truly decomposed into a few types of simple substructures or building blocks, great savings can be achieved. This is specially true for VLSI design where the chip comprises hundreds of thousands of components. Hence a good architecture is one that is highly pipelined and hierarchical and hence requires well structured, recursive algorithms.

Homogeneous machines are certainly easier to design. One merely patterns the layout for one of these processing elements and replicates this pattern appropriately. Such iterative layout patterns have been applied in the past for silicon memories, registers, and array multipliers. Similar techniques also have been used in constructing large software systems.

A number of high performance systolic processing architectures have appeared over the past two decades for performing various DSP algorithms. These are based on upon the original ideas of Kung [57] who defines a systolic architecture as being an array or network of Processing Elements (PEs), each capable of performing some simple operation (such as multiplication/accumulation) which synchronously computes and passes data through the system. Architectures such as these are particularly attractive for implementation with VLSI and wafer scale integration (WSI) technologies, owing to the associated simple and regular communication structures. The systolic array, which avoids
the classic bottleneck problem encountered with the Von Neumann computing device, is therefore very amenable to VLSI/WSI implementation, featuring the desirable properties of modularity, regularity, local interconnection and highly pipelined and synchronized multiprocessing.

The field of digital signal processing has experienced tremendous growth over the last two decades primarily due to dramatic improvements in Integrated Circuits (IC) process technology. Advances in device scaling have led to geometric increases in circuit densities with a resulting decrease in parasitic capacitance leading to higher speeds and lower power consumption. However, the technique of direct scaling is reaching a point of diminishing returns primarily due to problems encountered in providing inter chip and intra chip connections.

Many techniques are being pursued to find ways to further increase chip packaging densities. In the area of digital signal processing considerable demand exists for compact, high speed real time digital filters for use in radar communications and image processing. However, available real-time digital filters are often too slow, too costly, too complex or require too much power. In the past two decades the attention of some researchers has been directed to techniques for designing high speed digital filters making use of alternative number representation, or coding, schemes, compared to the traditional binary number system. A frequently cited example is the use of the Residue Number System (RNS) and associated arithmetic which allows modularity at the digit level, based on its algebraic properties.

1.2.1 Residue Number System

RNS is based on the principles of finite arithmetic, where operations are performed over integers. An RNS is composed of a collection of finite integer rings¹. Numbers are coded in each of the finite rings in an unweighted fashion. Because the rings are unweighted,

---

¹. A ring is a non-empty set, closed over addition, subtraction and multiplication [70].
addition and multiplication can be conducted simultaneously and in parallel in each of the rings without interaction. This results in highly parallel hardware design with characteristically high computational speed.

The use of computation over finite fields\(^1\)/rings is natural to communication researchers. Such computations form the backbone of many coding techniques in use today. The use of finite fields/rings operations for general computation has had some followers, but there has been a certain skepticism by a community that prefers the more natural weighted magnitude form of computations (e.g., binary and its several offshoots). For the special case of digital signal processing applications, where the inner product form represents a large computational burden on the processor, some researchers have turned their attention to finite field/ring processors as a possible alternative. A resurgence of interest was generated when large memories became available at relatively low cost. Multiplication turned out to be as simple as addition, if the computations were performed over finite rings, with the results being reassembled to a weighted magnitude form at the end. It now seems as though such computations also lend themselves admirably to the VLSI medium with dense modular and homogenous structures resulting. Some of the problems related to VLSI systems, such as clock skew, testing, and fault detection, may be mitigated via the use of this alternative computational medium.

From the VLSI implementation standpoint, the use of parallel computations over small finite rings removes the problems of clock skew across large connected two-dimensional bit-level systolic structures that occur in, for example, pipelined or carry-save adders with large bit lengths. The fact that clock skew is allowed to be present in a system means that skew itself can be used to advantage; for example, in the reduction of clock current spikes and thus in the reduction of system noise.

**Issues with the Residue Number System**

---

1. A ring is called a field when it is closed over division, that is the multiplicative inverse of the elements in the ring exist [70].
Residue number systems unfortunately suffer some drawbacks in spite of the many advantages mentioned earlier. The most significant is the dynamic range of operation. The dynamic range in a residue number system, is equivalent to the product of the relatively prime moduli chosen for the system (direct product ring). To increase the dynamic range one must add more moduli, or choose larger moduli. The relatively prime criterion for the moduli presents itself as a serious challenge, as it limits the choice for moduli. While certain moduli allow for the simplification of algorithms or reductions in hardware, other moduli do not offer such benefits. In order to have a sufficient dynamic range a designer may find it exceeding difficult to find moduli that share the same properties. Also by increasing in the number of moduli, or choosing larger moduli, the VLSI designs become more complex, and in some cases impractical. Much research has been done to suggest solutions to this dilemma. One solution, which is also the basis of this thesis work, is the Modulus Replication Residue Number System [111][112][115].

1.3 Modulus Replication Number System

The Modulus Replication Residue Number System (MRRNS) is different from classical residue number systems in that the computations are performed over direct product rings over the same modulus. This is achieved by encoding the digits of a weighted representation to polynomial residue rings, in one or more indeterminates. This encoding allows for the repeated use of the same moduli, hence the name “Modulus replication RNS”. A modulus in this system can be chosen based on it’s algebraic properties and the simplification that it allows for the hardware design. The desired dynamic range is obtained through replicated use of the same modulus in the encoding process. The resulting polynomial representation is not necessarily unique. It depends on what weighted representation is used to represent the numbers and also the number of indeterminates chosen for the polynomial representation. This redundancy leads to a number of encoding methods, with varying degrees of implementation complexity. The polynomial representation also effects the complexity of the computational hardware design. Finding an encoding method that is a fine balance between computational hardware and mapping complexity is at the heart of this thesis work.
1.4 Thesis Objectives

The work presented in this thesis has two main objectives. One is to devise a new mapping strategy for the Modulus Replication Residue Number system, that results in reduced hardware for inner-product based DSP applications, including a comparison study with other mapping methods. The second is to investigate the architectures and implementation requirements for the design of a DSP system based on the new mapping strategy and to provide a comparison between our proposed implementation and previous implementations. As part of the latter goal, a complete model of an FIR array described in Verilog Hardware Description Language (HDL) and fully simulated, is to provide proof of concept and design.

1.5 Thesis Organization

The organization of this thesis begins with the introduction (this chapter), followed by Chapter 2, which covers background material in finite arithmetic and a general treatment of the polynomial based mappings. Chapter 3 presents a detailed study of the MRRNS encoding, and investigates the effect of the number of indeterminates and the weighted number representations on the encoding process. It also presents an enhanced mapping technique that allows for more efficient hardware design using only a single modulus 257. Chapter 4 details a multiply-accumulator, namely the Fermat ALU, to be used with the enhanced MRRNS encoding and modifications made to the original architecture of the ALU are discussed. Chapter 5 provides verification of the reduced mapping theory by generating a floorplan (architecture) for the design of an FIR array, and using this architecture to fully simulate a 53-tap filter, described in Verilog, as an example of a practical implementation based on the encoding technique and Fermat ALU design. Finally, Chapter 6 presents conclusions and future work for this thesis.
Chapter 2

Number Theory for Digital Signal Processing

2.1 Introduction

Number theory, simply defined, is the study of the set of integers, and its related subsets and extensions, beyond their role as counting tools. The realm of number theory has a rich history which reaches as far back as ancient Babylonians, producing many notable mathematicians. Perhaps none left as great a mark on this research field as Gauss\(^1\), who helped change number theory from a collection of isolated problems to a coherent branch of mathematics.

As the majority of the field of Digital Signal Processing (DSP) deals with transformations that manipulate numbers in repetitious and recursive fashion, it would seem natural to look to number theoretic methods for solutions. Finite rings can offer considerable advantages over binary arithmetic in performing integer arithmetic. Most visibly is the use of finite rings to code integers as elements of a set of rings, with relatively prime moduli, allowing large dynamic range closed operations to be carried out by a set of parallel small ring calculations (Residue Number System).

\(^1\) Carl Friedrich Gauss (1777 - 1855)
This chapter presents some of the fundamentals of number theory and number representations[63][30][20][19][26], namely those that form the theoretical basis for this thesis work. More detailed information can be found in Section A on page 126. Any introductory books on Abstract Algebra and Number Theory will provide the proofs for the conjectures and theorems presented in this chapter. Where necessary, examples have been provided to further elucidate the use of concepts in relation to the thesis work. For a more in-depth study, the author suggests "Fundamentals of Number Theory" by William J. Leveque[63] as an excellent beginner's course and "Algebra" by Godement [30] for a more thorough and concise study.

2.2 Introductory Number Theory

Definition 2.1 A binary operation • on a set is a rule which assigns to each ordered pair of elements of the set some element of the set.

Definition 2.2 A binary operation • on a set $S$ is commutative if (and only if) $a \cdot b = b \cdot a$ for all $a, b \in S$. The operation • is associative if (and only if) $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ for all $a, b, c \in S$.

2.2.1 Groups

Definition 2.3 An algebraic system $\langle G, \cdot, e \rangle$ is called a group if $G$ is a set of elements on which a binary operation • is defined and $e$ is an element of $G$ so that the following conditions are satisfied for all $x, y, z \in G$.

1) $x \cdot y \in G$ (closure of $G$ under •)
2) $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ (associative law)
3) $x \cdot e = e \cdot x = x$ (identity element)
4) For every $x \in G$ there is a $y \in G$ such that $x \cdot y = y \cdot x = e$ (existence of inverse)
The element $e$ is called the identity of $G$, and it can be easily shown that $G$ contains no other element which satisfies (3) for every $x$ and that the inverse of every element is uniquely determined.

**Definition 2.4** A group is called Abelian if in addition to the properties (1) to (4) mentioned in the previous definition, we also have:

5) $x \cdot y = y \cdot x$ for every $x, y \in G$ (commutative law).

**Notation:** If $S$ is a set, then $|S|$ denotes the number of elements in $S$ (If $S$ is infinite then $|S|$ denotes the cardinal number of $S$ [27]).

**Definition 2.5** If $G$ is a group, then $|G|$ is called the order of the group.

**Definition 2.6** Let $H$ be a subgroup of a group $G$, and let $a \in G$. The left coset $aH$ of $H$ is the set $\{ ah | h \in H \}$. The right coset $Ha$ is similarly defined.

### 2.2.2 Isomorphism and Homomorphisms

**Definition 2.7** A function or mapping $\varphi$ from a set $A$ onto a set $B$ is a rule which assigns to each element $a$ of $A$ exactly one element $b$ of $B$. We say the $\varphi$ maps $a$ onto $b$, and that $\varphi$ maps $A$ onto $B$.

**Notation:** The classical notation to denote $\varphi$ maps $a$ into $b$ is $\varphi(a) = b$. Symbolic representation on the mapping is in the form $\varphi : A \rightarrow B$.

**Definition 2.8** An isomorphism of a group $G$ with a group $G'$ is a one-to-one function $\varphi$ mapping $G$ to $G'$ such that for all $x$ and $y$ in $G$: $\varphi(x \cdot y) = (\varphi x) \cdot (\varphi y)$. The two groups $G$ and $G'$ are isomorphic.

---

1. Named after the mathematician N.H. Abel (1802-1829)
What this means is that the two groups have the same structural features, and one can be made to look exactly like the other by a renaming of the elements.

**Definition 2.9** The mapping $\varphi: G_1 \rightarrow G_2$ is a homorphism from $G_1$ to $G_2$ if for every $x, y \in G$ we have $\varphi(xy) = (\varphi x) \cdot (\varphi y)$.

### 2.2.3 Cyclic Groups

**Definition 2.10** The cyclic group of finite order $n$ (denoted by $C_n$) is a group consisting of elements $e, g, g^2, \ldots, g^{n-1}$ with multiplication subject to the condition $g^n = e$ (Here, of course, $g^2 = g \cdot g$, $g^3 = g \cdot g \cdot g$, etc.).

An interesting derivation from the above is that any two cyclic groups of the same finite order are isomorphic. A group of elements having the property defined in this definition can be useful in performing multiplication. This can be done using index addition, similar to logarithms.

**Theorem 2.1** Every cyclic group is abelian.

**Definition 2.11** (Division Algorithm for $\mathbb{Z}$) Let $n$ be a fixed positive integer and let $h$ and $k$ be any integers. The number $r$ is defined such that

$$h+k=nq+r \quad \text{for } 0 \leq r < n$$

and is the sum of $h$ and $k$ modulo $n$. The notation is $h+k \equiv r(mod\ n)$ read $h+k$ is congruent to $r$ modulo $n$.

**Theorem 2.2** The set $\{0, 1, 2, \ldots, n-1\}$ is a cyclic group $\mathbb{Z}_n$ of $n$ elements under addition modulo $n$. 
**Theorem 2.3**  
Every group of prime order \( p \) is cyclic. In other words if \( p \) is prime, there exists an integer \( g \) (primitive root) for which \( \{g^n\}_p \) is a permutation of \( \{1, 2, \ldots, p-1\} \).

### 2.2.4 Rings and Fields

**Definition 2.12**  
\( <R, +, \cdot, 0, 1> \) is a commutative ring if

1. \( <R, +, 0> = R^+ \) is an Abelian group.
2. \( <R, \cdot> \) satisfies the closure, associative, and commutative axioms.
3. The distributive law holds, that is for every \( a, b, c \in R \):
   \[ a \cdot (b + c) = a \cdot b + a \cdot c \]

**Definition 2.13**  
A ring in which multiplication is commutative is a *commutative ring*.

A ring \( R \) with a multiplicative identity \( 1 \) such that \( a \cdot 1 = a \) for every \( a \in R \) is a ring with *unity*.

An element \( a \) of a ring \( R \) is said to be invertible if \( a \) has a multiplicative inverse in \( R \). That is, there is an element \( a^{-1} \) in \( R \) such that \( a^{-1} \cdot a = a \cdot a^{-1} = 1 \).

**Definition 2.14**  
Let \( R \) be a ring with unity. An element \( u \) in \( R \) is a unit of \( R \) if it has a multiplicative inverse in \( R \). If every non-zero element of \( R \) is a unit, then \( R \) is a *division ring*.

**Definition 2.15**  
Let \( m \) be a positive integer. We denote by \( R(m) \), the ring of integers modulo \( m \), i.e.

\[
R(m) = \{ S : \oplus_m, \ominus_m \}; S = \{ 0, 1, \ldots, m - 1 \}
\]

(2.1)

Where we use the notation \( a \oplus_m b \) and \( a \ominus_m b \) to imply the residue reduction of \( a \) and \( b \) modulo \( m \) with respect to addition and multiplication. We can extend the notion of addition and multiplication from the elements of \( S \) to all of the integers.
**Definition 2.16** If $a$ and $b$ are two nonzero elements of a ring $R$ such that $ab=0$, the $a$ and $b$ are divisors of zero.

**Theorem 2.4** In the ring $\mathbb{Z}_n$, the divisors of zero are precisely those elements which are not relatively prime to $n$.

Corollary: If $p$ is a prime, then $\mathbb{Z}_p$ has no divisors of zero.

**Definition 2.17** An isomorphism $\varphi$ of a ring $R$ with a ring $R'$ is a one-to-one function, mapping $R$ onto $R'$ such that for all $a, b \in R$:

1) $\varphi(a + b) = \varphi a + \varphi b$

2) $\varphi(a \cdot b) = \varphi a \cdot \varphi b$

The rings $R$ and $R'$ are then said to be isomorphic.

**Definition 2.18** A field $F=\langle F, +, \cdot, 0, 1 \rangle$ is a set of elements on which two binary operations $+$ and $\cdot$ are defined and containing two distinguished elements $0$ and $1$, with properties such that:

1) $\langle F, +, 0 \rangle = F^+$ (additive group) and $\langle F - \{0\}, \cdot, 1 \rangle = F^\times$ (multiplicative group) are Abelian groups.

2) The distributive law holds:

$$x \cdot (y + z) = (y + z) \cdot x = x \cdot y + x \cdot z$$

for all $x, y, z \in F$.

In other words a field is a commutative division ring.

**Definition 2.19** Two fields $F_1$ and $F_2$ are isomorphic if there is a mapping $\varphi$ which maps $F_1$ one to one onto $F_2$ and if for every $a, b \in F_1$ we have

$$\varphi(a + b) = \varphi(a) + \varphi(b) \quad \text{and} \quad \varphi(a \cdot b) = \varphi(a) \cdot \varphi(b)$$

If $F_1 = F_2$ then $\varphi$ is called an automorphism.
Definition 2.20

1) A field $F$ is of finite characteristic $p$, $[\text{char}(F)=p]$, if there is a least positive integer $p$ such that $(1+1+\ldots+1)=0$ in $F$ (addition of $p$ ones).

2) If there is no such integer, then $F$ is said to be of characteristic 0.

3) The characteristic of any finite field must be prime.

Theorem 2.5

The element $r$ in $\mathbb{Z}_m$ is invertible if and only if $r$ and $m$ are relatively prime in $\mathbb{Z}$. In particular when $p$ is a prime every element of $\mathbb{Z}_p$ except 0 is invertible.

From this theorem it follows that $\mathbb{Z}_p$ can be a field only if $p$ is prime.

Theorem 2.6

If $F$ is a finite field of characteristic $p$, then the additive group of $F$ is isomorphic to $(C_p)^r$, the direct product of $r$ copies of the cyclic group $C_p$. Consequently $|F| = p^r$ for some $r \geq 1$.

The most immediate result of this theorem is that if a finite field exists then its order must be a prime power.

Theorem 2.7

The multiplicative group of any finite field is cyclic.

Recalling that a group is cyclic if and only if all its elements can be expressed as powers of a single element, called the generator, in the case of a multiplicative group, this generator is called a primitive element.

In summary, the theory of finite fields turns out to be remarkably simple [15].

Any finite field has a prime power order $q=p^r$.

There is essentially just one field of order $q$.

The additive group of the field is $(C_p)^r$.

The multiplicative group of the field is $C_{q-1}$. 
The notation $F_q$ will be used for the unique (up to isomorphism) field of order $q$. These fields are often known as the Galois fields\(^1\), sometimes denoted by the symbol $GF(q)$.

**Theorem 2.8**  Let $q$ be an odd prime power. If $q \equiv 1 \pmod{4}$, then $-1$ has a square root in $F_q$ but if $q \equiv 3 \pmod{4}$, then $-1$ does not have a square root in $F_q$.

### 2.2.5 Direct Product Rings

If $R_1$ and $R_2$ are any two rings then we can define the cross-product ring $R_1 \times R_2$ as the set of pairs $\{s_1, s_2\} \in S_1 \times S_2$, with addition and multiplication defined component wise, as in Eqn. (2.2).

\[
(a_1, a_2) \oplus_{R_1 \times R_2} (b_1, b_2) = (a_1 \oplus_{R_1} b_1, a_2 \oplus_{R_2} b_2) \\
(a_1, a_2) \circ_{R_1 \times R_2} (b_1, b_2) = (a_1 \circ_{R_1} b_1, a_2 \circ_{R_2} b_2)
\]

(2.2)

An isomorphism between $R(M)$ and the direct product of $\{R(m_k)\}$ means that calculations over $R(M)$ can be effectively carried out over each $R(m_k)$, independently and in parallel. A final mapping to $R(M)$ is performed at the end of a chain of calculations. We have therefore broken down a calculation over a large dynamic range $M$. to a set of $L$ calculations over small dynamic ranges given by the $\{m_k\}$. This is shown to be the main advantage of using the RNS over a conventional weighted value numbering system (e.g. binary).

**Theorem 2.9**  If $m$ and $n$ are relatively prime, i.e., if the \texttt{gcd} of $m$ and $n$ is 1, then $\mathbb{Z}_m \times \mathbb{Z}_n$ is isomorphic to $\mathbb{Z}_{mn}$.

---

\(^1\) Named after Évariste Galois (1811-1832)
Corollary 1  If numbers $m_i$ for $i=1,\ldots, n$ are such that the $gcd$ of any two of them is equal to 1, then $\prod_{i=1}^{n} Z_{m_i}$ is cyclic and is isomorphic to $Z_{m_1 m_2 \ldots m_n}$.

2.2.6 Polynomial Rings

*Definition 2.21*  If $R$ is a commutative ring, then $R[x]=\{p(x)\}$ where $p(x)$ is an expression of the form $a_0 + \ldots + a_n x^n$, with $a_i \in R$, ($n$ is a positive integer), together with the usual addition and multiplication of the polynomial, is also a ring and is called the ring of polynomials in $x$ (indeterminate) with coefficients in $R$, and $p(x)$ is called a polynomial over the ring $R$. $a_n$ is called the leading coefficient of the polynomial. When $a_n=1$, the polynomial is called a monic polynomial. Polynomials of the form $a_0$ are known as constant polynomials. This definition can be extended to include multiple indeterminates.

The degree of a polynomial is the largest value of $n$ for which the coefficient of $x^n$ is not zero.

*Theorem 2.10*  Division Theorem for Polynomials

Let $F$ be a field and suppose that $a(x)$ and $b(x)$ are polynomials in $F[x]$, with $b(x) \neq 0$. Then there are unique polynomials $q(x)$ and $r(x)$ in $F[x]$ such that:

$$a(x)=b(x)q(x)+r(x) \quad (2.3)$$

where either $\deg r(x)<\deg b(x)$ or $r(x)$ is a zero polynomial.

*Definition 2.22*  We say that $g(x)$ is a divisor (or factor) of $f(x)$ in $F[x]$ if there is a polynomial $h(x)$ in $F[x]$ such that $f(x)=g(x)h(x)$. Given any two polynomials $a(x)$ and $b(x)$ in $F[x]$, we say that $d(x)$ is the greatest common divisor (gcd) of $a(x)$ and $b(x)$ if

1) $d(x)$ is a divisor of $a(x)$ and $b(x)$, and
2) any divisor of $a(x)$ and $b(x)$ is also a divisor of $d(x)$. 
**Theorem 2.11**  Let $F$ be a field and suppose $d(x)$ is a gcd of the polynomials $a(x)$ and $b(x)$ in $F[x]$. Then there are polynomials $\lambda(x)$ and $\mu(x)$ in $F[x]$, such that

$$d(x) = \lambda(x)a(x) + \mu(x)b(x)$$  \hspace{1cm} (2.4)

From the above theorem we can deduce analogous results to the factorization theorem for integers\(^1\) [15].

We define a polynomial $f(x)$ in $F[x]$ to be **irreducible** if it is not a constant polynomial and if $f(x) = g(x)h(x)$ in $F[x]$, then either $g(x)$ or $h(x)$ is a constant polynomial. Irreducible polynomials in $F[x]$ play the same role as primes in $\mathbb{Z}$.

**Theorem 2.12**  Any non-constant polynomial $f(x)$ in $F[x]$, can be expressed as a product of irreducible polynomials. If there are two such factorizations

$$f(x) = g_1(x), g_2(x), \ldots, g_r(x) = h_1(x), h_2(x), \ldots, h_s(x)$$

then $r=s$ and we can rearrange the order of factors so that $g_i(x)$ is a constant multiple of $h_i(x)$ ($1 \leq i \leq r$), that is, $g_i(x) = \alpha_i h_i(x)$ for some non-zero constant polynomial $\alpha_i$.

**Theorem 2.13**  **Factor Theorem**

Let $F$ be a field and suppose $f(x)$ is a polynomial in $F[x]$. Then $x - \alpha$ is a divisor of $f(x)$ in $F[x]$ if and only if $f(\alpha) = 0$ in $F$.

If $f(x)$ is a polynomial in $F[x]$ and $\alpha$ is an element of $F$, then we say the $\alpha$ is a root of the equation $f(x) = 0$ whenever $f(\alpha) = 0$.

**Theorem 2.14**  If $F$ is a field and $f(x)$ is a polynomial of degree $n \geq 1$ in $F[x]$, then the equation $f(x) = 0$ has at most $n$ roots in $F$.

---

1. This theorem states that every integer greater than or equal to two has a unique factorization into primes.
This theorem can be generalized for rings as:

**Lemma 1:** Let \( f(X) \) be a polynomial in \( X \) with coefficients in a commutative ring \( R \). Then an element \( a \in R \) is a root of \( f \) if and only if there exists a polynomial \( F(X) \) with coefficients in \( R \) such that \( f(X) = (X - a)F(X) \)

**Lemma 2:** Let \( f \) be polynomial in \( X \) with coefficients in \( \mathbb{Z}_M \). Suppose that \( f \) vanishes at points \( r_1, r_2, \ldots, r_m \in \mathbb{Z}_M \). If each difference \( r_i - r_j \), \( i \neq j \), is invertible in \( \mathbb{Z}_M \), then \( f \) is either the zero polynomial or it has a degree greater than or equal to \( m \). In other words

\[
f(X) \in \langle g(X) \rangle \text{ where } g(X) = \prod_{i=1}^{m} (X - r_i).
\]

Lemma 2 can be expanded to include multiple indeterminates.

**Corollary 2:** Let \( R \) be commutative ring with identity. For \( i=1,2,\ldots,n \) let \( X_1, X_2, \ldots, X_n \) be indeterminates over \( R \), let \( d_i \geq 1 \), let \( r_{i1}, r_{i2}, \ldots, r_{id_i} \in R \), and for each such \( i \) suppose that the set \( \{ r_{ij} - r_{ik} : 1 \leq j < k \leq d_i \} \) consists of invertible elements of \( R \). Let \( f(X_1, X_2, \ldots, X_n) \in R[X_1, X_2, \ldots, X_n] \) and suppose that \( f(r_{j_1}, r_{j_2}, \ldots, r_{j_{d_i}}) = 0 \) for each \( i=1,2,\ldots,n \) and each \( j_i=1,2,\ldots,d_i \). The \( f \) belongs to the ideal \( G \) generated by the polynomials \( g_i(X_j) = \prod_{j=1}^{d_i} (X_i - r_{ij}) \) for \( i=1,2,\ldots,n \).

**Definition 2.2.3** Evaluation map is the homomorphism that exists from \( R[X] \) to \( R \) given by fixing an element \( r \in R \) and then evaluating each polynomial \( f(X) \in R[X] \) at the value \( X=r \).
2.2.7 Ideals and Quotient Rings

**Definition 2.24** A subring \( N \) of a ring \( R \) satisfying \( rN \subseteq N \) and \( Nr \subseteq N \) for all \( r \in R \) is an ideal of \( R \).

**Definition 2.25** If \( N \) is an ideal in a ring \( R \), then the ring of cosets \( r+N \) under the induced operations is the quotient ring, or factor ring, or residue class ring of \( R \) modulo \( N \) and is denoted \( R/N \). The cosets are residue classes modulo \( N \).

**Definition 2.26** Polynomial Quotient Rings

Let \( g(X) \) be a polynomial of degree \( m \) from \( Z_M[X] \). Then \( Z_M[X]/(g(X)) \) is denoted the ring of polynomials which have coefficients in \( Z_M \) and have a degree less than \( m \). This ring is called the ring of quotients of \( Z_M[X] \) with respect to \( g(X) \).

**Isomorphisms and homorphisms**

**Theorem 2.15** Let \( R \) be a commutative ring with identity. For \( i = 1, 2, \ldots, n \) let \( X_1, X_2, \ldots, X_n \) be indeterminates over \( R \). Let \( d_i \geq 1 \), let \( r_{i1}, r_{i2}, \ldots, r_{id_i} \in R \), and for each such \( i \) suppose that the set \( \{ r_{ij} - r_{ik} : 1 \leq j < k \leq d_i \} \) consists of invertible elements of \( R \).

Let \( f(X_1, X_2, \ldots, X_n) \in R[X_1, X_2, \ldots, X_n] \) and suppose that \( f(r_{1j_1}, r_{2j_2}, \ldots, r_{nj_n}) = 0 \) for each \( i = 1, 2, \ldots, n \) and each \( j_i = 1, 2, \ldots, d_i \). Let ideal \( G \) be generated by the polynomials

\[
g_i(X_i) = \prod_{j=1}^{d_i} (X_i - r_{ij}), \quad i = 1, 2, \ldots, n.
\]

Then there exists an isomorphism between \( R[X_1, X_2, \ldots, X_n]/G \) and \( R \times R \times \ldots \times R \), where the direct product is taken \( \prod_{i=1}^{n} d_i \) times.

**Lemma 3:** Let \( M \) and \( d \) be positive integers. Then there exist distinct elements \( \{ r_1, r_2, \ldots, r_d \} \in Z_M \) such that each element of \( \{ r_i - r_j : 1 \leq i < j \leq d \} \) is invertible in \( Z_M \) if and only if each prime divisor \( p \) of \( M \) satisfies \( p \geq d \).
Theorem 2.16  Let \( M, n \), and \( d_1, d_2, \ldots, d_n \) be positive integers. Let \( X_1, X_2, \ldots, X_n \) be indeterminates over the ring \( \mathbb{Z}_M \). Suppose that for each prime divisor \( p \) of \( M \) we have \( p \geq d_i \) for \( i = 1, 2, \ldots, n \). Then for each such \( i \) there exists elements \( r_{i1}, r_{i2}, \ldots, r_{id_i} \in \mathbb{Z}_M \) such that each difference \( \{ r_i - r_j : 1 \leq i < j \leq d_i \} \) is invertible in \( \mathbb{Z}_M \). Let \( g_i(X_i) \in \mathbb{Z}_M[X_i] \) be the monic polynomial of degree \( d_i \) whose roots are \( \{ r_{ij} : 1 \leq j \leq d_i \} \). Let \( G \) be the ideal in \( \mathbb{Z}_M[X_1, X_2, \ldots, X_n] \) generated by the polynomials \( g_1(X_1), g_2(X_2), \ldots, g_n(X_n) \). Then there exists a ring isomorphism \( \mathbb{Z}_M[X_1, X_2, \ldots, X_n]/G \cong \mathbb{Z}_M \times \mathbb{Z}_M \times \cdots \times \mathbb{Z}_M \) where the direct product on the right is taken \( \prod_{i=1}^{n} d_i \).

2.3 Polynomial Based Mappings

The majority of signal processing and communication techniques, such as convolution, correlation, DFT computations, etc. are multiplication intensive, resulting in severe constraints on the data rate of real-time applications. This has led to the development of parallel architectures and faster number systems that result in higher throughputs, including polynomial representations. The common goal of these representations is to eliminate the need for partial product processing associated with polynomial multiplication. The intended application and the specifics of the polynomial mappings has led to three distinctive approaches as described in literature. The first uses polynomials to map the real and imaginary part of Gaussian integers and is known as Quadratic Residue Number System (QRNS) [40] [62]. The second approach which is a generalization of QRNS, uses polynomials to represent sequences of numbers and then performs partial product free multiplication to implement DSP applications like cyclic convolution and correlation [84][85][86]. The third approach which is used in this thesis work uses finite polynomials mapped from a weighted representation of a single data sample [111][112][115]. Here briefly the three different approaches are discussed and a comparison study is presented.
2.3.1 **Quadratic Residue Number System**

A method of handling complex data, so that the two channels for real and imaginary data are processed independently is to use the Quadratic Residue Number System [40] [62]. This method maps the real and imaginary data to two channels that compute over finite fields. The rings are built using Theorem A.6. on page 135.

For moduli of the form $4K+1$, -1 is a quadratic residue, therefore the monic quadratic $x^2 + 1 = 0$ has a solution in base ring $QR(m_i) = \{S: \oplus, \otimes\}$. If $j$ is a solution to the monic quadratic then $j$ and its multiplicative inverse will belong to $QR(m_i)$. Although an extension field cannot be built based on a solution of the quadratic, an extension ring can be generated. This ring is termed a quadratic ring. The extension element can be written as $AQ_i = (A_i^o, A_i^*)$ where $A_i^o = a_i \oplus j_i \otimes \alpha_i$ (normal) and $A_i^* = a_i \oplus -j_i \otimes \alpha_i$ (conjugate): $a_i, \alpha_i, A_i^o, A_i^* \in GF(m_i)$. This can be described in matrix form using the 2x2 Vandermonde matrix of the roots of $x^2 + 1 = 0$.

\[
\begin{bmatrix}
A_i \\
A_i^*
\end{bmatrix} = \begin{bmatrix}
1 & j_i \\
1 & -j_i
\end{bmatrix} \begin{bmatrix}
a_i \\
\alpha_i
\end{bmatrix}
\]  
(2.5)

The two binary operations of addition and multiplication, over the quadratic ring are computed as:

**Addition:**

\[
AQ_i \oplus (BQ_i) = (A_i \oplus B_i, A_i^* \oplus B_i^*)
\]  
(2.6)

**Multiplication:**

\[
AQ_i \otimes (BQ_i) = (A_i \otimes B_i, A_i^* \otimes B_i^*)
\]  
(2.7)
where the real and imaginary part of the product can be formed from the normal and conjugate parts of the result, \( Q \) and \( Q^* \) respectively, as:

\[
Y_{iR} = (2^{-1} \otimes_m (Q_i \oplus_m Q^*_i)) \\
Y_{il} = 2^{-1} \otimes_m j^{-1} \otimes_m (Q_i \oplus_m (-Q^*_i))
\]

(2.8)

This forms a commutative ring with identity. It should be noted that the ring is isomorphic to the finite ring of Gaussian integers, which will be denoted as \( R(m) \), and that both arithmetic operations only involve two base field operations (\( \oplus \), \( \otimes \)).

The concept can be extended to both special composite moduli and a system of quadratic rings using the direct sum mapping. The isomorphism given in Eqn. (2.9) is used to allow computations to be carried out in \( L \) parallel and smaller rings:

\[
R(M) = QR(m_1) \oplus QR(m_2) \oplus \ldots \oplus QR(m_L)
\]

(2.9)

\[
M = \prod_{i=1}^{L} m_i
\]

Proof of this can be found in [44].

Other variants and alternatives to QRNS are discussed in Appendix A "Properties of Number Systems" on page 126.

### 2.3.2 Polynomial Residue Number System

The Polynomial Residue Number system investigates the problem of multiplying two \( (N-l) \) degree polynomials modulo \( (x^N \pm 1) \) over some modular ring \( Z_p = \{0, 1, \ldots, p - 1\} \), where this ring is closed with respect to addition and multiplication modulo \( m \) [84] [86]. Applications of this number system are in the area of
filter design, linear and cyclic convolution and correlation. Generally, wherever the inner product of two sequences is required, this method can be used. It should be noted that the multiplication of two polynomials of order $N$ results in a polynomial of order $2N-1$. Hence in order for the product to belong to the same ring of polynomials as the inputs, the length of the input sequences should be chosen equal to $2N-1$, where the extra terms are set to zero. This is important when considering the PRNS method for the calculation of filter output sequences, or linear convolution. In the case of cyclic convolution, since the product is calculated modulo $(x^N \pm 1)$, there is no need to extend the input sequences [76].

The first step is to write the sequence as a polynomial:

$$A = a_0, a_1, \ldots, a_{N-1} \quad (2.10)$$

$$A(x) = a_0 + a_1x + \ldots + a_{N-1}x^{N-1}$$

The PRNS is based on the existence of an isomorphism between the quotient rings $Zp[x]/1x^N + 1$ and the ring $Z(p^N)$ for certain primes $p$ [119]. It can be proven that if $N$ divides $(p_i - 1)$ and $(p_i - 1)/2$, then the polynomials $x^N - 1$ and $x^N + 1$ have $N$ distinct roots in $Z(m)$, respectively, where $m = \prod p_i^{e_i}$ [84]. Hence the polynomial can be written as:

$$x^N \pm 1 \equiv (x - r_0)(x - r_1)\ldots(x - r_{N-1}) \quad (2.11)$$

$$r_i \in Z(m) \quad i = 0, 1, \ldots, N - 1$$

Then the product of two $(N-1)$st-degree polynomials modulo $(x^N \pm 1)$ in $Z(m)$ can be computed using only $N$ multiplications performed in parallel, and no additions. [119] offers a new interpretation of the complexity reduction achieved through PRNS, by providing a clearer understanding of the Chinese Remainder Theorem.
If \( P(m) \) is a finite structure containing the \((N-1)\)st-degree polynomials with coefficients in \( \mathbb{Z}(m) \), and if the polynomial \( x^N \pm 1 \) contains \( N \) distinct roots in \( \mathbb{Z}(m) \), then an isomorphic mapping \( f_N \) of \( P(m) \) onto \( \mathbb{Z}(m)^N = \mathbb{Z}(m) \times \mathbb{Z}(m) \times \ldots \times \mathbb{Z}(m) \) can be shown to exist, which is given by:

\[
 f_N : A(x) = a_0 + a_1 x + \ldots + a_{N-1} x^{N-1} \rightarrow A^*(x) = (a_0^*, a_1^*, \ldots, a_{N-1}^*) \quad (2.12)
\]

where \( a_i^* \) are calculated from the \( N \times N \) Vandermonde matrix as shown below:

\[
 \begin{bmatrix}
 a_0^* \\
 a_1^* \\
 \vdots \\
 a_{N-1}^*
\end{bmatrix}
 =
 \begin{bmatrix}
 1 & r_0 & \cdots & r_0^{N-1} \\
 1 & r_1 & \cdots & r_1^{N-1} \\
 \vdots & \vdots & \ddots & \vdots \\
 1 & r_N & \cdots & r_N^{N-1}
\end{bmatrix}
 \begin{bmatrix}
 a_0 \\
 a_1 \\
 \vdots \\
 a_{N-1}
\end{bmatrix}
 \quad (2.13)
\]

The reverse mapping is defined by:

\[
 f_N^{-1} : A^*(x) = (a_0^*, a_1^*, \ldots, a_{N-1}^*) \rightarrow A(x) = (a_0, a_1, \ldots, a_{N-1}) \quad (2.14)
\]

\[
 A(x) = \sum_{i=0}^{N-1} a_i^* Q_i(x) \quad Q(x) = N \left( 1 + r_i^{-1} x + r_i^{-2} x^2 + \ldots + r_i^{-(N-1)} x^{N-1} \right)
\]

where \( N^{-1} \) and \( r_i^{-1} \) are the multiplicative inverses of \( N \) and \( r_i \) in \( \mathbb{Z}(m) \). The rules of addition and multiplication in PRNS can be defined as:

**Addition:**

\[
 A(x) + B(x) \rightarrow A^*(x) + B^*(x) = (a_0^* \oplus_m b_0^*, a_1^* \oplus_m b_1^*, \ldots, a_{N-1}^* \oplus_m b_{N-1}^*)
\]
Multiplication:

\[ A(x) \cdot B(x) \rightarrow A^*(x) \cdot B^*(x) = (a^*_0 \otimes_m b^*_0, a^*_1 \otimes_m b^*_1, \ldots, a^*_{N-1} \otimes_m b^*_{N-1}) \]

From the multiplication rule, it can be seen that the product of two \((N-1)\)-degree polynomials can be performed with only \(N\) multiplication in \(\mathbb{Z}(m^N)\). The same polynomial would require \(N^2\) multiplications and \(N(N-1)\) addition performed in \(\mathbb{P}(m)\). The disadvantage of this method is the costly implementation of the forward and reverse mapping, which involves equating the polynomial for all the roots of \((x^N \pm 1)\) in the forward mapping, and equating Eqn. (2.14) on page 25 for the powers of the inverse roots. It should be noted that reduction in the multiplication complexity is at the cost of restricting the moduli set to allow \((x^N \pm 1)\) to have \(N\) distinct roots.

Skavantos et al. [85] use the PRNS method in computing linear convolution for complex numbers. A comparison is given with the QRNS method. They show that an \(N\)-point complex linear convolution can be computed with \(4N\) real multiplication using PRNS as opposed to \(2N^2\) in QRNS, excluding the requirements for forward and reverse mapping.

The limitation of the PRNS is that the size of the ring used is proportional to the size of the polynomial to be multiplied. Reference [85] offers a solution to this problem by employing 2-D PRNS techniques.

### 2.3.3 Moduli Replication RNS

The Moduli Replication RNS (MRRNS) technique [111][112][115], offers a new approach to obtaining the goal of increasing the dynamic range of operations without increasing the size of the moduli. This approach is based on representing weighted magnitude components (i.e. bits) of numbers into polynomial rings, allowing the replication of the moduli set. Original work done on this technique was based on by the
use of the ring of algebraic integers defined by a well chosen polynomial \( F(X) \) which is irreducible over the integers. However this revealed complex mapping strategies [23][28]. Where the previous polynomial mapping schemes were limited to a single indeterminate, MRRNS can allow for multiple indeterminates.

Figure 2.1 shows the encoding and decoding mappings required. For simplicity in the diagram the set of indeterminates \( X_1, X_2, \ldots, X_n \), equivalent to the set \( z^\alpha_1, z^\alpha_2, \ldots, z^\alpha_n \), have been represented symbolically as \( \mathbf{X} \) (a vector). The inputs are integers, though this could easily be extended to include complex numbers, by adding an extra indeterminate to represent the complex unit \( \mathbf{j} \).

**Figure 2.1 Rings and Homomorphisms of the MRRNS**

The result of the mapping is a direct product ring in which the computations now can be carried out independently over the same ring, at the end of which the inverse of the original mapping is applied. The output of the inverse mapping stage is a redundantly coded number, which is converted back to nonredundant representation using a combination of scaling and binary additions. Each stage of the mapping will be explained along with a simple example. Theorems that define the various isomorphisms/
homorphisms have already been discussed in Section 2.2.6 on page 17 and Section 2.2.7 on page 20.

The map $\Phi$ and $\Phi^{-1}$

The input data is assumed to have a fixed wordlength. This data is then represented as polynomials in $Z[X_1, ..., X_n]$. By representing the data as polynomials, instead of calculating the inner product of two sequences of integers, the inner product of two sequences of polynomials is performed and then the map $\Phi^{-1}$ is applied. This representation is not unique and hence the map $\Phi$ is not a homorphism. As well, there are several parameters that effect the mapping in this stage, namely polynomial order, data bit distribution, and the number of indeterminates. The trade-off among the various parameters is dictated by the dynamic range and the DSP computation (i.e. number of multiplications). The redundancy and trade-offs will be discussed in more detail in Chapter 3.

$\Phi^{-1}$ on the other hand is a homorphism, therefore it will preserve the sums and products, effectively preserving the inner product. It can be formally described by the following equation:

$$\Phi^{-1}: Z[X] \rightarrow Z$$

$$f(X) \rightarrow f(X = \{2^{\beta_0}, 2^{\beta_1}, ..., 2^{\beta_n}\})$$

The polynomial is reduced to an integer by replacing all the indeterminates in the answer polynomial with the respective powers of 2.

The map $\mu$ and $\mu^{-1}$
The modulus $M$ is selected in advance, usually for its algebraic properties. It can also be in the form of a composite modulus, in which case it factors into relatively prime moduli, $M = \prod_i m_i$. The polynomials in $Z[X]$ are then mapped into the ring $Z_M[X]$, by reducing the coefficients of the polynomial in $Z[X]$, modulo $M$. This map, however, can be trivial if the $X_i$ are chosen to be smaller than $M$. The formal definition of the mapping is:

$$\mu : Z[X] \rightarrow Z_M[X]$$

$$f(X) = \sum_{i_1 \in \{0, 1, \ldots, d_1\}, \ldots, i_n \in \{0, 1, \ldots, d_n\}} a_{i_1i_2\ldots i_n} X_1^{i_1}X_2^{i_2}\ldots X_n^{i_n}$$

$$\rightarrow f_M(X) = \sum_{i_1 \in \{0, 1, \ldots, d_1\}, \ldots, i_n \in \{0, 1, \ldots, d_n\}} [a_{i_1i_2\ldots i_n}]_M X_1^{i_1}X_2^{i_2}\ldots X_n^{i_n}$$

The map $\mu^{-1}$ must be carefully considered, since in general $\mu$ has no inverse. The coefficients of the answer polynomial must be their own residues (mod $M$), in order to avoid ambiguity and overflow error. The choice of $M$ is one of the parameters that can effectively reduce the chance of overflow. The issue of overflow analysis will be discussed in detail in Chapter 3.

The map $\Phi$ and $\Phi^{-1}$

The encoding continues with the map from $Z_M[X]$ to the direct product ring $\prod_i Z_{m_i}[X]$.

The map $\Phi$ further reduces the coefficients of the polynomials with respect to $\{m_i\}$. Formally this direct product ring is defined by:

$$\Phi : R(M) \rightarrow \prod_i R(m_i)$$

$$f_M(X) \rightarrow (f_{m_1}(X), f_{m_2}(X), \ldots, f_{m_k}(X))$$
Classic Chinese Remainder Theorem (CRT) assures that the map \( \Phi^{-1} \) exists and is an isomorphism between \( \prod_i Z_{m_i}[X] \) and \( Z_M[X] \), defined formally below:

\[
\Phi^{-1} : \prod_i R(m_i) \rightarrow R(M)
\]

\[
|a_{j_1 j_2 \ldots j_n}|_M = \left| \sum_i \hat{m}_i \left( \frac{1}{\hat{m}_i|_m} \right) |a_{j_1 j_2 \ldots j_n}|_M \right|
\]

Where \( j_1 \in \{0, 1, \ldots, d^*_{1}\}, \ldots, j_n \in \{0, 1, \ldots, d^*_{n}\} \) and \( d^* \) are the degrees of the output polynomial in each indeterminate. Figure 2.2 shows the embedded RNS within the MRRNS.

**Figure 2.2 Embedded RNS in MRRNS**

The map \( q \) and \( q^{-1} \)

The map \( q \) reduces the polynomials by calculating remainders using the Division algorithm (See "Division Theorem for Polynomials" on page 17.). Once again no
reduction will be necessary, since the ideals, \( \{g_i(X)\} \) are chosen such that they have a higher degree than the input polynomials. The map \( q \) is merely a formalism to give the existence of the isomorphism \( \Psi \).

\[
q : \mathbb{Z}_m[X] \rightarrow \mathbb{Z}_m[X]/(g_i(X))
\]

(2.19)

where \( i \in \{1, ..., L\} \)

Since the output polynomials have a different degree than that of the input polynomials, the degree of the ideals is chosen such as to handle the highest degree \( d^* \). In this manner the map \( q^{-1} \) exists and becomes trivial to compute.

*The map \( \Psi \) and \( \Psi^{-1} \)*

This final map is the evaluation map, which will evaluate all polynomials at all possible roots of the ideal using the Vandermonde matrix [30]. The final direct product ring will contain \( d_1d_2...d_n \) individual copies of each of the rings \( \mathbb{Z}_{m_i}^* \); \( d_i \) being the degree of \( g_i(X) \).

Since \( \Psi \) is an isomorphism, \( \Psi^{-1} \) poses no problem. The inverse of the Vandermonde matrix used in \( \Psi \) is used to perform this map.

*The inner product*

Once the data is mapped to the final direct product ring, the inner product computations can take place over \( d_1d_2...d_n \) replicated channels. Once all the computations are performed, the result from each channel goes through the reverse mapping process to yield the integer answer.

*Illustrative Example*
As an example, let us consider multiplying 13 by 15, for \( M=17 \), using a single indeterminate \( X=2^2=4 \). Input polynomials representing 13 and 15 will be of degree 1, so the output polynomial will be a degree two polynomial. The ideal will be chosen so that it has a higher degree than the output, \( g(X)=X(X-1)(X+1) \).

The first step is to write the inputs as polynomials (map \( \phi \))

\[
13 = 3X + 1 \\
15 = 3X + 3
\]  

The next step is to reduce the coefficients of the polynomials by \( M \). Since the coefficients are already smaller than \( M \), no reduction is necessary. Also, since \( M \) is prime and not a composite modulus, the next mapping stage is also eliminated.

The final stage requires a mapping of the polynomials to a direct product ring, using the Vandermonde matrix. This matrix of all possible roots of the ideal is shown below:

\[
\begin{bmatrix}
0^0 & 0^1 & 0^2 \\
1^0 & 1^1 & 1^2 \\
-1^0 & -1^1 & -1^2
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 \\
1 & 1 & 1 \\
1 & -1 & 1
\end{bmatrix}
\]  

The multiplication of the matrix by the coefficients of the input polynomial will yield the values passed to 3 replication channels to perform the multiplication operation. For the polynomial representing 13 the result is:

\[
\begin{bmatrix}
1 & 0 & 0 \\
1 & 1 & 1 \\
1 & -1 & 1
\end{bmatrix} \otimes_{17} \begin{bmatrix}
1 \\
3 \\
0
\end{bmatrix} = \begin{bmatrix}
1 \\
4 \\
15
\end{bmatrix}
\]  

Similarly for the polynomial representing 15 the result is:
\[
\begin{bmatrix}
1 & 0 & 0 \\
1 & 1 & 1 \\
1 & -1 & 1
\end{bmatrix} \otimes_{17} \begin{bmatrix}
3 \\
3 \\
0
\end{bmatrix} = \begin{bmatrix}
3 \\
6 \\
0
\end{bmatrix}
\tag{2.23}
\]

Now the multiplication can be performed independently in each replicated channel, the result being:

\[
\begin{bmatrix}
1 \otimes_{17} 3 \\
4 \otimes_{17} 6 \\
15 \otimes_{17} 0
\end{bmatrix} = \begin{bmatrix}
3 \\
7 \\
0
\end{bmatrix}
\tag{2.24}
\]

Now that the computations in the replicated path have been completed, we proceed with the inverse mapping from the direct product ring to the polynomial ring. Here we use the inverse of the Vandermonde matrix to obtain the result polynomial.

\[
\begin{bmatrix}
1 & 0 & 0 \\
1 & 1 & 1 \\
1 & -1 & 1
\end{bmatrix}^{-1} \otimes_{17} \begin{bmatrix}
3 \\
7 \\
0
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 \\
0 & 9 & 8 \\
0 & 16 & 9
\end{bmatrix} \otimes_{17} \begin{bmatrix}
3 \\
7 \\
0
\end{bmatrix} = \begin{bmatrix}
3 \\
12 \\
9
\end{bmatrix}
\tag{2.25}
\]

Finally we reconstruct the integer value from the polynomial representation by replacing the indeterminate with 4.

\[
9X^2 + 12X + 9 = 9 \times 16 + 12 \times 4 + 9 = 201
\tag{2.26}
\]

Clearly this result is incorrect as \(13 \times 15 = 195\). The reason for the error is the overflow that occurred in second computational channel \((4 \times 6 = 24 > 17)\), when the result of the operation exceeded \(M\), and so was reduced to 7. Had we chosen to multiply 13 by itself, using the same parameters, no overflow would have occurred and the correct result would have been decoded. The point of this example is to show that the parameter of a MRRNS system, such as indeterminate, \(M\) must be carefully selected by performing an error analysis of the system to be implemented, to minimize the chance of overflow.
2.4 Polynomial mapping comparisons

Comparisons of the three different polynomial mapping approaches, highlighting their advantages and disadvantages are presented below. The study clearly shows that of the three polynomial mappings discussed, MRRNS show definite superiority by removing the traditional RNS restriction of relatively prime moduli to achieve large dynamic ranges. The polynomial encoding is simple and lends itself very well to VLSI implementation. The decoding can be further simplified by using only a single modulus, and hence eliminate the need for the cumbersome CRT, which the other two methods can not avoid. In fact the ability to perform the computation over replications of a single modulus as opposed to composite modulus, is the driving force behind this thesis work.

PRNS:

Disadvantages:

The limitation of the PRNS is that the size of the ring used is proportional to the size of the polynomial to be multiplied. Reference [85] offers a solution to this problem by employing 2-D PRNS techniques. A major difficulty in this approach is to decide what it is that the polynomials should represent. After all a signal seldom arrives in the form of a polynomial package, with the exception of complex signals. Other problems lie in the quantization of the data as well as restrictions imposed on the prime divisors, $p$, of the modulus, $M$, since all the roots of the polynomial $X^n \pm 1$ are assumed to be in $Z_M$.

Advantages:

PRNS technique allows an increase in the dynamic range, without a corresponding increase in the modulus $M$. Skavantos et al [85] use the PRNS method in computing linear convolution for complex numbers. A comparison is given with the QRNS method. They show that an $N$-point complex linear convolution can be computed with $4N$ real
multiplication using PRNS as opposed to \(2N^2\) in QRNS, excluding the requirements for forward and reverse mapping.

**QRNS:**

**Disadvantages:**

QRNS imposes yet another condition on the moduli (that they be of the form \(4k+1\)) and thus creates the need for larger moduli, leading to complications in VLSI implementation of such systems where any inherent advantages of finite ring versus binary computations may be lost. QRNS also suffers from the same problem as classical RNS, i.e. in order to have a large dynamic range, a large number of moduli is needed.

**Advantages:**

Use of the QRNS can simplify the design and decrease area. The choice of the moduli in the form \(4k+1\) allow for the roots of \(x^2 + 1 = 0\) to be elements of the field, in other words -1 is a quadratic residue. This offers an advantage over the traditional complex residue number system, by reducing the number of multiplications in a complex multiply operation from 4 to 2.

**MRRNS:**

**Disadvantages:**

The disadvantage of this technique is the large redundancy in the finite ring representation of the data; however, this is compensated for by the repeated use of small moduli. As well, the additional hardware does not increase the complexity of design. An alternate mapping strategy is developed in [113], where the data are written as polynomials in several variables, each representing a different power of 2. This will, in effect, increase the
dynamic range of computation, although it will also increase the redundancy of computational hardware.

Advantages[115]:

1. There are no quantization problems. The data, either real or complex are assumed to be of a given fixed wordlength. No approximations or scaling are used in encoding the data.

2. The polynomials used are of a general nature, so that no restrictions are placed on the prime divisors of the moduli, except in the case of a QRNS representation of complex data, in which case the condition is the usual one of \( p=4k+1 \) for prime divisors \( p \) of the modulus \( M \).

3. The same small moduli can be used many times, which allows VLSI implementations of systems which can process data of a large wordlength, using direct products of many copies of modular rings with small moduli.

4. Encoding is a simple matter of diverting the bits of the input data to the proper channels. Decoding is only complicated in so far as the Chinese remainder theorem is used, and even then only for a limited number of small moduli. Scaling if used in decoding, is simplified by the ring structures used; certain monomials can be ignored as they represent insignificant digits.

2.5 Summary

This chapter has presented an introduction to number theory. It has covered preliminary concepts of finite arithmetic, along with definitions and theorems that form the foundations of discrete mathematics, allowing for the construction of suitable number systems for specific algorithms. This is achieved by mapping numbers from one number system to another, resulting in a suitable computational environment which may compensate for the additional hardware overhead needed to implement the number conversion. Algebraic structures such as quotient ring of polynomials and direct product
rings have also been discussed in this chapter. We have also reviewed mathematical tools and concepts necessary for such number conversions. Finally, several different polynomial mappings, cited in the literature, have been summarized and compared.
Chapter 3

Redundant Polynomial Mapping in MRRNS

3.1 Introduction

The advantages of computing over direct product rings are well documented in the literature [93]. From a VLSI standpoint these advantages include reduction in clock skew [48], natural fault tolerance [49] and ease of testability [50]. From a computation perspective, such techniques promise high speed arithmetic by allowing totally independent computations, each over a small dynamic range.

Conventional finite ring mapping is based on the Residue Number System (RNS) [93], where the inputs, data and coefficients, are mapped to the residues via a modulo reduction operator for each moduli. The operations are then performed component wise for residues of like moduli. The final result is mapped back using the Chinese Remainder Theorem (CRT). The dynamic range is limited to the product of the moduli. The MRRNS mapping [111], on the other hand, allows replication of the computational modulus while still allowing a large computational dynamic range. The choice of a single computational modulus as opposed to a composite one, is highly desirable, as it eliminates the complicated CRT operation in the reverse mapping process. A new mapping scheme will be presented in this chapter that exploits the redundancies in the polyno-
mial and binary representation, resulting in suitable hardware/power/speed trade-offs for a variety of DSP applications, while minimizing the probability of overflow error.

3.2 Modulus Replication

As mentioned in Chapter 2, the polynomial mapping in MRRNS begins by representing integers as polynomials. Since the application for this system is digital signal processing, the integer values are represented in binary form, which can be construed as a special case of polynomial mapping where a single indeterminate \( X=2 \) is used and the coefficients of the polynomial are the bits of the binary representation.

Assuming a binary representation of the positive integer data, \( s \):

\[
s = \sum_{i=0}^{B-1} s_i 2^i
\]  

(3.1)

where \( s_i \in \{0, 1\} \). A negative number is represented similarly except that \( s_i \in \{0, -1\} \).

The following theorem allows us to rewrite Eqn. (3.1) using a polynomial of the form given in Eqn. (3.2):

\[
s = \sum_{i_1=0}^{d_1} \sum_{i_2=0}^{d_2} \cdots \sum_{i_n=0}^{d_n} s_{i_1,i_2,...,i_n} 2^{(i_1 \beta_1 + i_2 \beta_2 + \cdots + i_n \beta_n)}
\]  

(3.2)

where \( B = \beta_0 > \beta_1 > \cdots > \beta_n = \beta \).

Theorem 3.1 Let \( B \leq \beta \prod_{i=1}^{n} (1 + d_i) \). Then any integer \( s \) (or its negative) lying in the range \([-2^B + 1, 2^B - 1]\) has a representation (Eqn. (3.2)) where \( 0 \leq s_{i_1,i_2,...,i_n} \leq 2^B - 1 \) and \( X_i (i=1, 2, ..., n) \) are to be evaluated by \( X_i = 2^\beta_i \).
The $d_i$ are the degrees of the polynomial in the indeterminate $X_i = 2^i$.

Proof:

See reference [17].

The polynomial representation is now mapped to a finite polynomial ring (modulo $M$) and then to a direct product ring, $Z_M \times Z_M \times \ldots \times Z_M$, using an evaluation map; this is implemented by multiplying the finite polynomial ring coefficient vector by a Vandermonde matrix of all the possible roots of the mapping ideal. [111]. From Theorem 2.14, it can be deduced that if $M$ is prime all elements of $R(M)$ can be roots of ideal, removing restrictions on the choice of the ideal.

We can now perform independent calculations over each of the copies of $Z_M$. Conditions for reversing the mapping procedure are discussed in [111] and Section 2.3.3. An important restriction is that the resulting finite polynomial ring coefficients do not exceed the modulus, $M$, during the computations. and applications using both very small moduli [111] and large moduli [48] have been investigated in literature. In the latter case the Fermat ALU was introduced for inner product processing in which the ring modulus was $M = 257 \times 17$; both prime factors being Fermat primes. As mentioned earlier, $M$ is chosen primarily for its algebraic properties, which can effect the architecture and implementation of a MRRNS system.

Exclusive of $M$, there are still several other design variables to consider when using MRRNS, namely the number of indeterminates, the size of the indeterminates, the input wordlengths, and the blocklength of the inner product. These variables are not independent from one another and they are ultimately chosen to avoid (or at least reduce the probability of) any overflow errors in the output polynomial coefficients. In most DSP applications the blocklength of the inner product is fixed, so the objective is to manipulate
the other parameters, to achieve a system that allows for the implementation of the DSP
application.

3.3 Modular Overflow Error of Inner Products

Prior to implementing a DSP system with MRRNS, an analysis of the probability of error
occurring in the output polynomial is conducted. Errors occur when coefficient
computations overflow the ring modulus, M. To demonstrate this problem, we look at a
typical multiplication of polynomials, \( A(X_1, \ldots, X_n) = \sum_{j_1=0}^{d_{s_1}} \cdots \sum_{j_n=0}^{d_{s_n}} a_{j_1} \cdots a_{j_n} X_1^{j_1} \cdots X_n^{j_n} \)
and \( B(X_1, \ldots, X_n) = \sum_{i_1=0}^{d_{b_1}} \cdots \sum_{i_n=0}^{d_{b_n}} b_{i_1} \cdots b_{i_n} X_1^{i_1} \cdots X_n^{i_n} \) in the MRRNS:

\[
C(X) = A(X) \cdot B(X) = \sum_{k_1=0}^{d_{s_1}+d_{b_1}} \cdots \sum_{k_n=0}^{d_{s_n}+d_{b_n}} c_{k_1 \ldots k_n} X_1^{k_1} \cdots X_n^{k_n}
\]  

(3.3)

with:

\[
c_{k_1 \ldots k_n} = \sum_{\forall i_1 + j_1 = k_1} \cdots \sum_{\forall i_n + j_n = k_n} a_{j_1} \cdots a_{j_n} b_{i_1} \cdots b_{i_n}
\]

(3.4)

Overflow will occur when:

\[
|c_{k_1 \ldots k_n}| > \frac{M-1}{2} - 1
\]

(3.5)

In an RNS (or other type of integer) system, overflow occurs in the output dynamic range
whereas the overflow discussed in this system is when one or more of the output
polynomial coefficients exceeds the computational modulus. In an RNS system, overflow
of the dynamic range is absolutely not tolerated as it results in ambiguity, whereas in a

---

1. We have an empirical probability error limit of 0.05% for an acceptable filter performance [111].

Redundant Polynomial Mapping in MRRNS  Modular Overflow Error of Inner Products  41
The MRRNS system we can allow for a small amount of overflow without compromising our final integer result. This is a consequence of the way the coefficients in the output polynomial map to the final dynamic range. When reconstructing the final integer value from the polynomial, a lower positioned coefficient overflow will typically cause an error in the lower insignificant bit positions of the final dynamic range. By the same token, should higher positioned coefficients experience an overflow, a significant error may occur in the final result. The mapping of the full computational dynamic range of a polynomial multiplication to a 50% reduced output dynamic range is shown in Figure 3.1 (we also show an overflow condition in the middle coefficient).

**Figure 3.1 Integer map from polynomial coefficients**

Example 3.1 Assume $A(X)$ and $B(X)$ are both 9 bits, with $X=8$, and $M=257$. The output dynamic range of the multiplication will be 18 bits. Since $A(X)$ and $B(X)$ will both be degree 2 polynomials, the output polynomial will be degree 4, so the total dynamic range of the polynomial representation will be:

$$\log_2([X^4 + X^3 + X^2 + X + 1] \times 257) \text{ bits}$$

For our assumed indeterminate mapping, the coefficient in the output polynomial, with most growth and hence the most probability of error is $c_3$. The overflow will be some integer multiple of 257 and the error will be of the form:
\[ |c_3 - c_3|X^2 = 257 \times 2^5 \equiv 2^{11} \] (3.7)

Since the total dynamic range of computation is 21 bits, this error will not be very significant. The fractional uncertainty [104] will be 0.0976 and the computed result in the case of an overflow will be equal to the "true value" ±0.0976.

From Eqn. (3.3) - Eqn. (3.5) it can be seen that the size and the degrees of the input polynomial coefficients, contribute to the number and the size of the terms in Eqn. (3.5). The size of the input polynomial coefficients is governed by the choice of the indeterminates, which in turn effects the degree of the input polynomial. In other words the problem of minimizing the overflow error is ultimately controlled by the input polynomial representation.

There is a double redundancy present in this representation: the signed digit redundancy of Eqn. (3.1) and the polynomial redundancy due to multiple indeterminates of Eqn. (3.2). The objective this chapter is to exploit both forms of redundancy to design a more efficient system.

3.4 Polynomial Representation

The polynomial representation can be categorized into two types: Single indeterminates and multiple indeterminates. We will investigate the merits and drawbacks of each type in this section. For both types of representation, the assumption is made that a single modulus, \( M \), is being used.

3.4.1 Single Indeterminate

The single indeterminate polynomial representation is a special case of Eqn. (3.2), where only one indeterminate \( X = 2^B \), is employed, resulting in a single level mapping of the binary number to polynomial coefficients.
Assume that a positive input number (integer) $0 \leq s \leq 2^B - 1$ has $B$ bits (excluding the sign bit): this number can be written in the form:

$$s = \sum_{i=0}^{B-1} s_i 2^i = \sum_{k=0}^{\lceil (B-1)/\beta \rceil - 1} \left( \sum_{j=0}^{\beta - 1} s_{(\beta k + j)} 2^j \right) (2^\beta)^k = \sum_{k=0}^{\lceil (B-1)/\beta \rceil - 1} c_k (2^\beta)^k$$  \hspace{1cm} (3.8)

where $c_k \in \{0, 1, \ldots, X - 1\}$. The same is also true if the number is a negative integer. In this case all the $c_k$ are negative. It is worth noting that for a single indeterminate, the polynomial representation in Eqn. (3.8) is unique, and the degree of the polynomial is $\lceil (B-1)/\beta \rceil - 1$. Since the DSP computation to be performed are inner products, the output polynomial will have a degree equal to $2 \times \lceil (B-1)/\beta \rceil - 1$ resulting in an ideal with degree $2 \times \lceil (B-1)/\beta \rceil - 1$. For simplicity an ideal of the form shown below will be chosen:

$$g(X) = X^{\lceil (B-1)/\beta \rceil - 1} \prod_{i=1}^{\lceil (B-1)/\beta \rceil - 1} (X - r_i)(X + r_i)$$  \hspace{1cm} (3.9)

with the roots of the ideal being 0 and $\pm r_i$ where $r_i = \{1, \ldots, \lceil (B-1)/\beta \rceil - 1\}$.

From a hardware implementation point of view, there will be $2 \times \lceil (B-1)/\beta \rceil - 1$ replicated computational channels.

**Inner Product**

The inner product of two polynomials\(^1\) $A(X) = \sum_{i=0}^{d_a} a_i X^i$, and $B(X) = \sum_{j=0}^{d_b} b_j X^j$

represented with a single indeterminate can be written as:

---

1. Assume the polynomials $A$ and $B$ represent $B_a$ and $B_b$ bit integers, respectively. The degree of each polynomial will be $\lceil (B_a - 1)/\beta \rceil$ and $\lceil (B_b - 1)/\beta \rceil$. 

---
\[ C(X) = A(X) \cdot B(X) = \sum_{k} c_k X^k \]  

(3.10)

where

\[ c_k = \sum_{i + j = k} a_i b_j \]  

(3.11)

The coefficients with the greatest number of summands are:

\[ c_{\lfloor (d_a + d_b)/2 \rfloor} \]  

for \((d_a + d_b)\) an even number

\[ c_{\lfloor (d_a + d_b)/2 \rfloor} \text{ and } c_{\lfloor (d_a + d_b)/2 \rfloor + 1} \]  

for \((d_a + d_b)\) an odd number

with \(\left\lfloor \frac{d_a + d_b}{2} \right\rfloor + 1\) terms and a maximum magnitude of \(\left\lfloor \frac{d_a + d_b}{2} \right\rfloor + 1 \times (X - 1)^2\). For an inner product of blocklength \(N\), this value will increase to \(\left\lfloor \frac{d_a + d_b}{2} \right\rfloor + 1 \times (X - 1)^2 \times N\).

Overflow will occur when:

\[ \left( \left\lfloor \frac{d_a + d_b}{2} \right\rfloor + 1 \right) \cdot (X - 1)^2 \cdot N > \frac{M - 1}{2} - 1. \]  

(3.12)

For many custom \(DSP\) applications, the bitlength of the data and coefficient stream, along with the blocklength of the computation, is predetermined and fixed. So the only variables affecting the overflow error are the indeterminate and \(M\). Figure 3.2 shows a plot of the \(LHS\) of Eqn. (3.12), against \(X\), for input polynomials of \(8 \leq B \leq 20\) bits.
Figure 3.2 Plot of LHS of Eqn. (3.12), against $X$ and $B$

The graph above shows that regardless of the input bitlength an increase in the size of the indeterminate results in a larger magnitude for the coefficient with the largest sum of terms. A pessimistic $L1$ norm\(^1\) would determine that for indeterminates equal to or greater than 8, only a single inner product can be performed as seen in Figure 3.3.

---

1. Given a vector $x=[x_1, x_2, \ldots, x_n]$, the $L1$ norm is a vector norm. denoted $\|x\|_1$, is defined as

$$\|x\|_1 = \sum_{r=1}^{n} |x_r|$$
However, this is the worst case scenario. For a more practical approach we need to perform an analysis based on the distribution of the input data streams, to determine the probability of error. This will be detailed further in Section 3.6.1.

### 3.4.2 Multiple Indeterminates

Assume a positive binary representation of $B$ bit integer data (including sign bit):

$$s = \sum_{i=0}^{B-2} s_i 2^i$$  \hspace{1cm} (3.13)

where $s_i \in (0, 1)$. $B$ bit integers. $s \in \mathbb{Z}$ can be represented as elements of the ring $\mathbb{Z}[X_1, X_2, \ldots, X_n]$, where $[X_1, X_2, \ldots, X_n] = [2^{\beta_1}, 2^{\beta_2}, \ldots, 2^{\beta_n}]$. This representation is highly redundant. If we expand $s$ in powers of $2^{\beta_i}$, we may rewrite Eqn. (3.13) as:

$$s = \sum_{i_1 = 0}^{d_1} s_{i_1} 2^{i_1 \beta_i}$$  \hspace{1cm} (3.14)

1. $\beta_0 = B - 1$ and $\beta_n = \beta$
where $0 \leq s_{i_1} \leq 2^{\beta_1} - 1$. Any integer $0 \leq s \leq 2^{\beta - 1} - 1$ can be represented in this form, given the assumption that $d_1 + 1 \geq \beta_0 / \beta_1$. Next we expand the coefficient $s_{i_1}$ in terms of $2^{\beta_2}$ resulting in:

$$s_{i_1} = \sum_{i_1 = 0}^{d_2} s_{i_1} 2^{i_1 \beta_2} \quad (3.15)$$

The assumption here is that $d_2 + 1 \geq \beta_1 / \beta_2$ to ensure that $0 \leq s_{i_1} \leq 2^{\beta_1} - 1$. In a similar fashion we expand all resulting coefficients, arriving at:

$$s = \sum_{i_1 = 0}^{d_1} \sum_{i_2 = 0}^{d_2} \cdots \sum_{i_n = 0}^{d_n} s_{i_1, i_2, \ldots, i_n} 2^{(i_1 \beta_1 + i_2 \beta_2 + \cdots + i_n \beta_n)} \quad (3.16)$$

where $0 \leq s_{i_1, i_2, \ldots, i_n} \leq 2^\beta - 1$ with the requirement that $d_n + 1 \geq \beta_{n-1} / \beta_n$. This exercise can be repeated for a negative integer number, where all the coefficients will be negative. To arrive at an expression for the ideal, we make the assumption that the degrees of the two input streams, for the DSP application, are of degree $d^a_i$ and $d^b_i$ in the variable $X_i$.

We choose the ideals $g_i(X_i)$ of degree $d_i = d^a_i + d^b_i$. in the form of:

$$g_i(X_i) = \prod_{j=1}^{d_i} (X_i - r_{ij}) \quad (3.17)$$

As in the single indeterminate case, consecutive integers are chosen for the $r_{ij}$ such that

for $d_i$ odd \quad $r_{i1} = (1 - d_i)/2$ and $r_{i, j + 1} = r_{ij}$ for $1 \leq j \leq d_i - 1$

for $d_i$ even \quad $r_{i1} = 1 - d_i/2$ and $r_{i, j + 1} = r_{ij}$ for $1 \leq j \leq d_i - 1$. 
The number of replicated computational channels will be \( \prod d_i \).

The redundancy using multiple indeterminates, results in polynomials of varying degrees and coefficient sizes. The ramifications are that, unlike the single indeterminate case, the polynomial representation is not unique. Several factors effect the level of redundancy, namely, the number of indeterminates, the size of the indeterminate, and finally, the integer value to represent. The following example will demonstrate these observations.

**Example 3.2** Assume \( s=137 \) is represented in binary with 8 bits. The polynomial representation for this number, given two indeterminate \( X=2^2 \) and \( Y=2^4 \) is \( s=10001001 \), and the representation of the binary weight in terms of the indeterminates is shown in Table 3.1.

<table>
<thead>
<tr>
<th>( s )</th>
<th>( \text{in } X )</th>
<th>( \text{in } Y )</th>
<th>( \text{in } X \text{ and } Y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( 2^0 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>( 2^1 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>( 2^2 )</td>
<td>( X )</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>( 2^3 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>( 2^4 )</td>
<td>( X^2 )</td>
<td>( Y )</td>
</tr>
<tr>
<td>0</td>
<td>( 2^5 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>( 2^6 )</td>
<td>( X^3 )</td>
<td>( XY )</td>
</tr>
<tr>
<td>1</td>
<td>( 2^7 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As a result, based on Eqn. (3.16) and its constraints, the number can be represented as shown in Eqn. (3.18).

\[
s = 2XY + 2X + 1 \tag{3.18}
\]
Excluding the single indeterminate representation, \( s \) has only one representation in \( X \) and \( Y \). From the table we can see that certain bit positions have multiple representations; however, the number we chose to represent does not have “ones” in those bit positions, and therefore yields a single “multiple indeterminate” representation. On the other hand for \( s = 11010010 \), since there are “ones” in the fifth and seventh bit, it can be represented as:

\[
\begin{align*}
3XY + X^2 + 2 \\
3XY + Y + 2 \\
3X^3 + Y + 2
\end{align*}
\]  

(3.19)

From Table 3.1 on page 49, it can be deduced that for an 8-bit integer, with indeterminates \( X = 2^2 \) and \( Y = 2^4 \), there can be at most three different representations, with a fourth representation being a single indeterminate representation in \( X \).

If we repeat this example with a different set of indeterminates, \( X = 2^2 \) and \( Y = 2^3 \), the table of the binary weights will be of the form:

<table>
<thead>
<tr>
<th>( s )</th>
<th>in ( X )</th>
<th>in ( Y )</th>
<th>in ( X ) and ( Y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2^0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2^1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2^2</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2^3</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2^4</td>
<td>X^2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2^5</td>
<td></td>
<td>XY</td>
</tr>
<tr>
<td>0</td>
<td>2^6</td>
<td>X^3</td>
<td>Y^2</td>
</tr>
<tr>
<td>1</td>
<td>2^7</td>
<td></td>
<td>X^2Y</td>
</tr>
</tbody>
</table>

In this case the maximum number of representations for an 8-bit integer will be 14.
Inner Product

The coefficients of the inner product of two numbers (blocklength $N$) represented in multiple indeterminates is given by Eqn. (3.4) and repeated again here:

$$c_{k_1 \ldots k_n} = \sum_{\forall i_1 + j_1 = k_1} \ldots \sum_{\forall i_n + j_n = k_n} a_{i_1 \ldots i_n} b_{j_1 \ldots j_n}$$  \hspace{1cm} (3.20)

From Theorem 3.1. we arrive at the conditions to ensure a correct representation following the inverse map:

$$|c_{k_1 \ldots k_n}| \leq N(2^B - 1)^2 (1 + \min(d^a_1, d^b_1)) \ldots (1 + \min(d^a_n, d^b_n))$$  \hspace{1cm} (3.21)

since $|c_{k_1 \ldots k_n}| > \frac{M - 1}{2} - 1$ causes an overflow error. then the condition for choosing $M$ will be:

$$\frac{M - 1}{2} > N(2^B - 1)^2 \prod_{i=1}^{n} (1 + \min(d^a_i, d^b_i)) + 1$$  \hspace{1cm} (3.22)

3.4.3 Comparisons

In the single indeterminate case, the degree of the ideal is determined by the bitlength of the input data and the indeterminate. $\lceil((B - 1)/\beta)\rceil + 2$. The coefficients of the polynomial can be easily determined in a unique fashion using Eqn. (3.8). The range of the coefficients will be $-(X - 1) \leq c_i \leq X - 1$, and the size of the Vandermonde matrix will be $((B - 1)/\beta) \rceil + 2) \times ((B - 1)/\beta) \rceil + 2).$ The overflow is dictated by the degree of the polynomial and the size of the coefficients. On the other hand adding more indeterminates may help reduce the degree of the polynomial and also the range of the coefficients, but it will also increase the size of the Vandermonde matrix. This is particularly evident when more than 2 indeterminates are used and the increase in the hardware for the additional replicated channels cancels out any advantages that may have been gained due to smaller coefficient size. The reduction in coefficient size and
polynomial degree will help reduce the probability of overflow; however, there is a heavy price to pay in terms of hardware overhead. Also, a disadvantage is that there is no fixed method for mapping an integer to a polynomial, and preprocessing is required to determine the best form of polynomial representation.

The best compromise is to use a single indeterminate system, but with reduced coefficient sizes, similar to that achieved for a multiple indeterminate case. In the next section we will look at the different ways of representing the binary integer data as a means of achieving this goal.

3.5 Binary Representation

3.5.1 Unsigned binary number

An unsigned binary number represents only the magnitude of the integer number. As data in DSP applications can be both positive and negative, the sign of the number is important information, that is not forwarded in this number system. Generally this number system is not used for computation unless there is an apriori knowledge that all the data will be of the same sign.

3.5.2 Signed binary number

A signed binary number can be used to represent both positive and negative values by using the most-significant bit (called the sign bit), which takes on a value of 0 or 1 for '+' or '-', respectively. The remaining bits contain the absolute magnitude.

In a signed (or sign and magnitude) binary representation, the range of numbers covered by $B$ bits is:

$$[-(2^{B-1} - 1), (2^{B-1} - 1)]$$  \hspace{1cm} (3.23)
A number can be uniquely represented, though technically there can be two representations for zero (+0 and -0).

**Polynomial Representation**

The magnitude of a $B$ bit integer (including sign bit) represented in this manner can be mapped to a polynomial in indeterminate $X = 2^B$ by Eqn. (3.8), resulting in the magnitude of the polynomial coefficients ($cm_i$). The sign bit (MSB) of the integer is then added as the MSB of the coefficients of the polynomial, $c_i$, hence converting them to a signed representation.

$$s = \sum_{i=0}^{B-1} s_i 2^i = \sum_k c_m (2^\beta)^k$$  \hspace{1cm} (3.24)

where:

$$c_k = s_{B-1} 2^\beta + 1 + cm_k$$  \hspace{1cm} (3.25)

**Figure 3.4 Polynomial mapping for sign and magnitude representation**

The polynomial representation is unique since there are no redundancies in the binary representation of the integer. The magnitude of the polynomial coefficients can be obtained simply by segmenting the magnitude of the integer data into $\beta$ bits. The
coefficients are then represented by $\beta + 1$ bits by adding the sign bit from the integer data as the MSB of the coefficient representation. The hardware requirements are minimal, and are essentially wire connections from the integer bits to the polynomial coefficient bits, as shown in Figure 3.4.

3.5.3 One’s complement

In the one’s complement representation, the leftmost bit is 0 for positive numbers and is 1 for negative numbers, as it is for the signed magnitude representation. A negation is made by complementing the bits. The one’s complement representation is not commonly used. This is at least partly due to the difficulty in making comparisons when there are two representations for 0. There is also additional complexity involved in adding numbers. The range of numbers covered by this representation with $B$ bits is:

$$[-(2^{B-1} - 1), (2^{B-1} - 1)]$$

(3.26)

Polynomial Representation

There are two possibilities for the polynomial mapping. The first is that the integer data is already in a one’s complement representation, and polynomial coefficients in one’s complement need to be produced. The second is that the integer data is in a sign and magnitude representation and that polynomial coefficients in one’s complement are desired.

In the first case, since the magnitude of the integer number is not obvious from this representation. Eqn. (3.8) cannot be used to directly map the integer to a polynomial. Instead the one’s complement representation needs to be converted to a sign and a magnitude representation, and then mapped to a polynomial. Following the mapping, the resulting coefficients need to be converted back to one’s complement. The hardware requirements, compared to a sign and magnitude representation, are inverters for forward and reverse one’s complement mapping. The polynomial representation of an integer
represented as one's complement is unique, following the same argument made for the
sign and magnitude representation. The polynomial mapping is identical to Figure 3.4.

In the second case, the need for the initial conversion of the integer data back to a sign and
magnitude representation is removed. The only hardware required is to convert the sign
and magnitude represented polynomial coefficients to a one's complement representation.
In both these cases the coefficients will still be in the range of \([-X+1, X-1]\), represented by
\(\beta\) bits for the magnitude and one bit for the sign. The polynomial mapping is shown in
Figure 3.5.

3.5.4 Two's complement

In the two's complement representation, the leftmost bit is 0 for positive numbers and is 1
for negative numbers. Negation is formed by adding 1 to the one's complement negation.
There is only one representation of 0 for this format in which all the bits of the number are
zero. There is an equal number of positive and negative numbers. Zero is considered to be
a positive number because its sign bit is 0. The range of numbers covered by this
representation with \(B\) bits is:

\[
[-2^{B-1}, 2^{B-1} - 1]
\]  

Polynomial Representation

The same considerations as in the one's complement case can be made here, with the
exception that more hardware is required for the forward and inverse two's complement
mapping. In this representation the polynomial coefficients will be in the range of \([-X, X-1]\),
represented by \(\beta\) bits for the magnitude and one bit for sign.
3.5.5 Signed digit representation [74]

The signed digit representation and its properties were introduced by Avizienis [5]. As the name indicates, this representation uses both positive and negative digits; hence the digit set $D$ is defined as $D = \{\bar{\beta}, \ldots, \bar{1}, 0, 1, \ldots, \alpha\}$. A symmetric signed digit representation is one where $\alpha = \bar{\beta}$, in which case the following notation is used to define the digit set:

$$D_{(b, a)} = \{\bar{\alpha}, \ldots, \bar{1}, 0, 1, \ldots, \alpha\} \quad (3.28)$$

where $b$ is the radix and $\alpha$ is the greatest digit in the digit set. There are certain taxonomy that is associated with signed digit representation. If $\alpha < \frac{b-1}{2}$, the digit set is said to be incomplete, as some numbers cannot be represented. If $b$ is odd and the number of digits is equal to $b$, i.e. $\alpha = \frac{b-1}{2}$, then the digit set is said to be complete, but not redundant (case
for conventional number representation). Finally if 
\[ \alpha \geq \left\lceil \frac{b}{2} \right\rceil, \]
the digit set is redundant. 

Additionally if \[ \alpha = \left\lfloor \frac{b}{2} \right\rfloor, \]
then the digit set is minimally redundant, and if \[ \alpha = b - 1 \]
then the digit set is maximally redundant, and finally if \[ \alpha > b - 1 \]
the digit set is over-redundant.

The radix 2 signed digit representation \( D_{(2, 1)} \) is called the Signed-Binary Digit (SBD) or Redundant-Signed Digit (RSD) representation. In this representation a number \( x \) is defined as:

\[
x_{(2, 1)} = \sum_{i} s_i 2^i
\]

(3.29)

where \( s_i = D_{(2, 1)} = \{ \bar{1}, 0, 1 \} \). The range of number that can be covered by this representation is:

\[
[-(2^B - 1), 2^B - 1]
\]

(3.30)

The signed digit representation of the number with the fewest number of non-zero bits is called a canonic signed digit representation (CSD). Properties of CSD are that no two consecutive digits are non-zero and that the representation is unique [74].

**Polynomial Representation**

Again there are two possibilities in terms of the polynomial mapping hardware. The first is that the integer data is already in a signed digit representation and maps to polynomial coefficients with signed digit representation. The second is that the integer is in a sign and magnitude representation and polynomial coefficients in signed digit are desired.
In the first case, to map $B$ bit data represented as signed digit to a polynomial in indeterminate $X = 2^B$, we simply use Eqn. (3.8) with the only modification that the upper bound of the summation will now be $B$. This is owing to the fact that no sign bit exists and the sign of the number is encoded into the representation. As a result the mapped polynomial will be of degree $\lceil B/\beta \rceil - 1$ with $\beta$ bit coefficients. As the integer data can be represented redundantly in this representation, the polynomial map will also be redundant. The degrees of the various polynomial maps will be the same, but their differences will be in the size of the coefficients. The mapping for this case is shown in Figure 3.6.

**Figure 3.6 Polynomial mapping for signed digit (case 1)**

In the second case Eqn. (3.8) is used to determine the coefficients in a sign and magnitude representation of $\beta + 1$ bits, then a conversion process is performed to map these coefficients to their signed digit representation with the same number of bits. This $\beta + 1$ bit representation can be expanded as follows:

$$c_k = \sum_{j=0}^{\beta} s_j 2^j = s_{\beta} 2^\beta + \sum_{j=0}^{\beta-1} s_j 2^j = s_{\beta} X + \sum_{j=0}^{\beta-1} s_j 2^j$$

(3.31)

where $s_j \in \{-1, 0, 1\}$.
The most significant bit of each coefficient is in the same bit position as the *LSB* of the next higher coefficient value. As a result the *MSB* that results from the each coefficient conversion will be added to the next higher positioned coefficient. A sequential order of conversion is required, starting from the least significant coefficient, to produce \( \beta \) bit coefficients.

**Figure 3.7 Polynomial mapping for signed digit (case 2)**

Clearly, the redundant representation that consistently yields a polynomial with the smallest coefficients, will be the desired representation for our system. Also it should be noted that the number of input bits can be increased, while still representing the same input data range, to reduce the coefficient sizes of the polynomial. In this case the resulting polynomial may be several degrees higher than \( \lceil B/\beta \rceil \) and the ramifications in terms of the number of replicated channels needs to be investigated. This new form of polynomial representation using signed digit representation will be called the *Enhanced Polynomial Representation* and will be detailed in Section 3.6.
Table 3.3 Coefficient representations for X=8

<table>
<thead>
<tr>
<th>coefficient integer value</th>
<th>signed binary representation</th>
<th>signed digit representations</th>
</tr>
</thead>
<tbody>
<tr>
<td>-7</td>
<td>1111</td>
<td>1001 (CSD) 0111</td>
</tr>
<tr>
<td>-6</td>
<td>1110</td>
<td>1010 (CSD) 0110</td>
</tr>
<tr>
<td>-5</td>
<td>1101</td>
<td>1011 (CSD) 0101 (CSD) 0111</td>
</tr>
<tr>
<td>-4</td>
<td>1100</td>
<td>0100 (CSD) 1100</td>
</tr>
<tr>
<td>-3</td>
<td>1011</td>
<td>0011 (CSD) 1101 (CSD) 0101 (CSD)</td>
</tr>
<tr>
<td>-2</td>
<td>1010</td>
<td>0010 (CSD) 1110</td>
</tr>
<tr>
<td>-1</td>
<td>1001</td>
<td>0001 (CSD) 1111 (CSD) 0111 (CSD)</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>0001 (CSD) 1111 (CSD) 0111</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>0010 (CSD) 1110</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>0011 (CSD) 1101 (CSD) 0101 (CSD)</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>0100 (CSD) 1000</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>1011 (CSD) 0100 (CSD) 0101 (CSD)</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>1010 (CSD) 0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>1001 (CSD) 0111</td>
</tr>
</tbody>
</table>

Table 3.3 shows the range of the coefficients for X=8, along with the sign and magnitude representation and the redundant signed digit representations.

3.5.6 Comparison

Compared to the range represented by sign and magnitude representation, the signed digit representation range is twice as large for the same number of bits. Also the signed digit representation does not have the restriction that the coefficients of the polynomial map always be $-(X-1) \leq c_i \leq X-1$, as all the other binary representations previously discussed. This makes signed digit representation the obvious choice for reducing the coefficient size while maintaining the same mapping hardware.
3.6 Enhanced Polynomial Representation

Having investigated the different forms of polynomial representations, we conclude that the best polynomial map can be reached by using a single indeterminate with a signed digit representation. The former allows us to fix the input data segmentation into coefficients, and the latter allows us to obtain multiple polynomial representation in which the coefficient sizes are different. The objective is to find a signed digit representation that consistently produces smaller coefficient sizes.

For a chosen indeterminate \( X = 2^\beta \) the simplest polynomial mapping of the binary data is to group the data into \( \beta \)-bit segments, with each segment representing a coefficient. The sign of the coefficients is the same as the sign of the input number and the range covered is \([-X+1, X-1]\) with \( \beta + 1 \) bits.

The range \([-X+1, X-1]\) can also be represented redundantly using a signed digit representation with \( \beta + 1 \) digits. If we expand this range, as shown in Table 3.4, we see that a signed digit representation exists in which the \( \beta \) LSBs of the representation are equal to or less than \([X/2]\) in magnitude. The MSB bit from the representation is then added to the next significant coefficient. The new significant coefficient is examined and converted to a signed digit representation with the same property of having the \( \beta \) LSB equal or less than \( X/2 \). This continues until all the coefficients have been converted. The result is a polynomial with coefficients that are now in the range of \([-X/2, X/2]\).

This reduction in the coefficient sizes will also reduce the coefficient dynamic range by up to 75% for a single multiplication. Although this may increases the degree of the polynomial, which clearly increases the hardware, the input polynomial will at most be one degree higher than the simple mapping method with the MS coefficient of only \( \pm 1 \). The trivial map of the binary number to a polynomial, in the signed digit polynomial mapping scheme (case 1), is replaced with a very small coefficient lookup table in the enhanced polynomial mapping. The succeeding mapping stages will not change in spite of

Reducant Polynomial Mapping in MRRNS   Enhanced Polynomial Representation
a possible increase in the degree of the polynomial. The effect of the extra coefficient is realized by a polynomial addition, the results of which will be summed with the output in the final addition stage, as will be shown later.

Table 3.4 Signed digit representation of the range \([-X+1, X-1]\)

<table>
<thead>
<tr>
<th>integer number</th>
<th>β+1 bit signed digit representation</th>
<th>MSB</th>
<th>βLSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-X/2)</td>
<td>(-X+1)</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>(-X/2)</td>
<td>(-X+2)</td>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>(\vdots)</td>
<td>(\vdots)</td>
<td>(\vdots)</td>
<td>(\vdots)</td>
</tr>
<tr>
<td>(-X/2-1)</td>
<td>(-X+X/2-1)</td>
<td>-1</td>
<td>(X/2-1)</td>
</tr>
<tr>
<td>(-X/2)</td>
<td>(-X+X/2)</td>
<td>(-X/2)</td>
<td>0 (-X/2)</td>
</tr>
<tr>
<td>(-X/2+1)</td>
<td>(-X/2+1)</td>
<td>0</td>
<td>(-X/2+1)</td>
</tr>
<tr>
<td>(\vdots)</td>
<td>(\vdots)</td>
<td>0</td>
<td>(\vdots)</td>
</tr>
<tr>
<td>(X/2-X/2-1)</td>
<td>(-1)</td>
<td>0</td>
<td>(-1)</td>
</tr>
<tr>
<td>(X/2-X/2=0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(X/2-X/2+1)</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(\vdots)</td>
<td>(\vdots)</td>
<td>0</td>
<td>(\vdots)</td>
</tr>
<tr>
<td>(X/2-1)</td>
<td>(X/2-1)</td>
<td>0</td>
<td>(X/2-1)</td>
</tr>
<tr>
<td>(X/2)</td>
<td>(X-X/2)</td>
<td>(X/2)</td>
<td>0 (X/2)</td>
</tr>
<tr>
<td>(X/2+1)</td>
<td>(X-X/2+1)</td>
<td>1</td>
<td>(-X/2+1)</td>
</tr>
<tr>
<td>(X/2+2)</td>
<td>(X-X/2+2)</td>
<td>1</td>
<td>(-X/2+2)</td>
</tr>
<tr>
<td>(\vdots)</td>
<td>(\vdots)</td>
<td>1</td>
<td>(\vdots)</td>
</tr>
<tr>
<td>(X/2+X/2-2)</td>
<td>(X-2)</td>
<td>1</td>
<td>(-2)</td>
</tr>
<tr>
<td>(X/2+X/2-1)</td>
<td>(X-1)</td>
<td>1</td>
<td>(-1)</td>
</tr>
</tbody>
</table>

The procedure in converting the coefficients to the smaller range is presented below.

Let us represent the input polynomial using the simple map as in Eqn. (3.32):

\[
c_nX^n + \ldots + c_2X^2 + c_1X^1 + c_0X^0
\]

(3.32)
For values of $|c_i| < \frac{X}{2}$, we do nothing; for values of $|c_i| > \frac{X}{2}$, we can reduce the magnitude of $c_i$ by incrementing the next highest coefficient as follows:

$$
c_i' = \begin{cases} 
X - c_i & (c_i > X/2) \\
X + c_i & (c_i < -X/2)
\end{cases}
$$

$$
c_{i+1}' = \begin{cases} 
c_i + 1 & (c_i > X/2) \\
c_i - 1 & (c_i < -X/2)
\end{cases}
$$

(3.33)

where we assume that the indeterminate, $X$, is replaced by its appropriate power of 2. This may, unfortunately, require a non-zero $n+1$ order component in the polynomial representation of Eqn. (3.32). It is unfortunate because a product of $n+1$ degree polynomials will overflow the $2n+1$th degree ideal in the original computation. With the enhanced mapping, the input data polynomial may be of degree $n+1$ and so the product polynomial will be of maximum degree $2(n+1)$

But this multiplication can be represented in terms of a summation of, at most, $2(n+1)$ degree polynomials, and an $n$ degree polynomial product. If we let the original data representation be $A(X) = c_nX^n + \ldots + c_1X^1 + c_0X^0$ and $B(X) = cb_nX^n + \ldots + cb_1X^1 + cb_0X^0$. the enhanced polynomial representations will be:

$$
ca'_{n+1}X^{n+1} + ca'_{n}X^{n} + \ldots + ca'_{1}X^1 + ca'_{0}X^0 = ca'_{n+1}X^{n+1} + A'(X)
$$

$$
cb'_{n+1}X^{n+1} + cb'_{n}X^{n} + \ldots + cb'_{1}X^1 + cb'_{0}X^0 = cb'_{n+1}X^{n+1} + B'(X)
$$

(3.34)

then we can write the modified product after the enhanced mapping as:

$$
(c'a'_{n+1}X^{n+1} + A'(X))(cb'_{n+1}X^{n+1} + B'(X)) =

ca'_{n+1}cb'_{n+1}X^{2(n+1)} + ca'_{n+1}X^{(n+1)} \cdot B'(X) +

\quad cb'_{n+1}X^{(n+1)} \cdot A'(X) + A'(X) \cdot B'(X)
$$

(3.35)

1. A $2n+1$th order ideal allows the multiplication of $2n$ order polynomials without overflow.
where \( a'_{n+1}, c b'_{n+1} \in \{-1, 0, 1\} \). The first three terms are simply polynomial additions which can be performed outside of the direct product ring with minimal hardware. The last term is a polynomial multiplication of two \( n \) degree polynomials, which can be performed over the an \( 2n+1 \) degree ideal, over the direct product ring. This allows us to use the same number of replication channels for degree \( n+1 \) polynomial computation, as we would for degree \( n \) polynomial computations, while increasing the dynamic range of the computation.

The conversion process is performed in sequence, starting from the ‘least significant’ coefficient: the ROMs store the following function:

\[
\text{for } \left( |c_i| < \frac{X}{2} \right) \quad c'_i = c_i \quad \text{Cout}_i = 0
\]

\[
\text{for } \left( |c_i| > \frac{X}{2} \right) \quad c'_i = \pm X + c_i \quad \text{Cout}_i = \pm 1
\]

The adders will sum the \( \text{Cout}_{i-1} \) from the previous stage with the input coefficients \( a_i \) from the next stage, where the number of stages is to equal to the number of original polynomial coefficients.

The following example will demonstrate the merits of the enhanced polynomial map compared to other mapping schemes.

**Example 3.3** Assume an integer equal to 309 with a magnitude of 9 bits. If we represent this number in sign digit form and then map to a polynomial in \( X=8 \), the possible representations are as follows:

The integer represented in sign and magnitude is 0100110101. The polynomial coefficients are: \( c_0=5, c_1=6, c_2=4 \). Alternatively this number can be represented in signed digit form, using Eqn. (3.5), as:
1. 01001110\(\overline{\text{T}}\), with \(c_0=-3, c_I=7, c_2=4\), conversion on \(c_0\)

2. 01010\(\overline{\text{T}}\)0101, with \(c_0=5, c_I=-2, c_2=5\), conversion on \(c_I\)

3. 1\(\overline{\text{T}}\)00110101, with \(c_0=5, c_I=6, c_2=-4, c_3=1\), conversion on \(c_2\)

4. 1\(\overline{\text{T}}\)001110\(\overline{\text{T}}\), with \(c_0=-3, c_I=7, c_2=-4, c_3=1\), conversion on \(c_0, c_2\)

5. 010100\(\overline{\text{T}}\)0\(\overline{\text{T}}\), with \(c_0=-3, c_I=-1, c_2=5\), conversion on \(c_0, c_I\)

6. 10\(\overline{\text{T}}\)00\(\overline{\text{T}}\)0\(\overline{\text{T}}\), with \(c_0=-3, c_I=-1, c_2=-3, c_3=1\), conversion on \(c_0, c_I, c_2\)

7. 10\(\overline{\text{T}}\)0\(\overline{\text{T}}\)0101, with \(c_0=5, c_I=-2, c_2=-3, c_3=1\), conversion on \(c_I, c_2\)

The various maps are obtained by converting one or more of the coefficients from the sign and magnitude representation to a signed digit representation in the range of \([-X/2, X/2]\). Mapping 6, above, demonstrates this conversion being performed on all the coefficients, and clearly has the best overall coefficient sizes. This conversion ensures that the magnitude of the coefficients does not exceed 4, or \(X/2\), unlike the other seven redundant representations.

To further investigate the merits of mapping 6, we will perform a random set of polynomial multiplications with each mapping and examine the output polynomial coefficients. The input values will be random numbers, and we will use the Extend simulation package to perform the simulation. The plot of the output coefficients can be seen in Figure 3.8 - 3.15 (note that the y-axis scale is different for each plot). The simulation has been performed on 1000 random samples. Of interest is the range of the output coefficients.
Figure 3.11 Map 3

Figure 3.12 Map 4

Figure 3.13 Map 5
As the graphs above show, the smallest output range following a multiplication belongs to the enhanced polynomial map (Map 6).

3.6.1 Overflow Error Analysis

The experiment shown in Example 3.3 can be performed repeatedly to generate an "intuitive" measure of the likelihood of an overflow occurring; however, ultimately it is desirable to come up with a fixed number that represents this probability of error through means of a statistical analysis.

In order to model this error, we need to first model the distribution of the input streams. For most DSP applications one of the streams represents the coefficients and the other the data. We will model these separately since they are distributed differently. Probability
Generating Functions (PGFs) are used to describe the two input streams. The Probability Generating Function or "Moment Generating Function" is defined in Eqn. (3.37) [32]:

\[ M(x) = E[e^x] \]  
(3.37)

where \( X \) is a random variable and \( M \) is its associated moment generating function. Important properties of the moment generating functions are:

- Under *mild* conditions, the generating function completely determines the distribution.
- The moment generating function of the sum of two random variables is the product of their moment generating functions.
- The moment generating function of \( N \) independent equally distributed random variables is the probability generating function to the \( N \)th power.

**Input Data Stream**

A uniform distribution, as shown in Figure 3.16 is assumed for the input data stream of \( B \) bits.

![Figure 3.16 Uniform Distribution of Input Data](image)

Distributions for the polynomial coefficients of the input data with the indeterminate \( X=2^B \), will also be uniform. From the distributions, expressions for the PGF of each polynomial coefficient \( c_n \), of \( \beta+l \) bits, before applying the enhanced mapping (see Section 3.6), will be:
\[
\text{PGF}(c_i) = \frac{1}{2^{\beta + 1}} \quad \text{for} \quad (-X + 1 \leq c_i \leq X - 1)
\]

(3.38)

After the enhanced mapping application, the distributions of the polynomial coefficients \(c_i\)' will no longer be uniform.

The \text{PGFs} will be:

\[
\text{PGF}(c_i') = \begin{cases} 
\frac{1}{2^\beta} & \text{for} \left( -\frac{X}{2} + 1 \leq c_i' \leq \frac{X}{2} - 1 \right) \\
\frac{1}{2^{\beta + 1}} & \text{for} \quad c_i' \in \left\{ -\frac{X}{2}, \frac{X}{2} \right\}
\end{cases}
\]

(3.39)

**Input Filter Coefficient Stream**

The input filter coefficient stream can be modelled in the same fashion as the input data stream. However, upon further examination of typical filter coefficients, it becomes evident that their distribution is anything but uniform.

To derive a model for the coefficient stream distribution, a large number of low-pass, band-pass and high-pass filters were studied. By plotting an accumulated histogram of the coefficients of all the filters in our study, we can arrive at a distribution model. Figure 3.17 shows such a plot for over 200 filters with filter blocklength=53. The coefficients have been normalized to the range \([-512, 511]\) (10 bits including the sign).
Clearly, most of the filter coefficients lie within less than 20% of the full range. We now perform the polynomial mappings, which result in separate histograms for each polynomial coefficient.

The *PGF* describing the distribution of the filter coefficients is shown in Eqn. (3.40):

\[
PGF(x) = \frac{N(x)}{L}; \quad -(2^{B-1} + 1) \leq x \leq 2^{B-1} - 1
\]  

(3.40)

where \( L \) is the number of filters and \( B \) is the bitlength for the coefficients. The filter coefficients are then mapped to polynomials with and without the enhanced mapping scheme. New histograms for the polynomial coefficients are next produced, and from these histograms, distribution profiles for the polynomial coefficients are found.

Figure 3.18 and Figure 3.19 show the resulting polynomial coefficient histograms, with and without enhanced polynomial mapping, based on the experiment shown in Figure 3.17 for a degree 2 polynomial with indeterminate \( X=8 \).

The *PGF* for each polynomial coefficient \( c_i \), without the enhanced mapping, is calculated from the histogram plot, as shown in Eqn. (3.41):
\[
PGF(c_i) = \frac{N(X^i)}{L \times TAP} \quad -(X - 1) \leq c_i \leq (X - 1) 
\] (3.41)

Similarly for the enhanced mapping scheme, the \textit{PGF} definition of the polynomial coefficients, \(c_i'\), is calculated from its histogram plots using eqn. (3.41), except that \(-\frac{X}{2} \leq c_i' \leq \frac{X}{2}\).

**Figure 3.18 Histogram of polynomial coefficients without enhanced mapping**

In order to calculate the probability of error for a given indeterminate, input data bitlength and the inner product block length, we have modified the program \textit{MODULUS} [111]. The original program assumed a sign and magnitude representation for the input data and used the mapping technique described in Section 3.5.2 to produce the polynomial coefficients. It also only accepted normal and uniform distributions for the input streams. The modifications extend the mapping procedures to the enhanced polynomial mapping...
scheme and also allow custom distributions for the input data (i.e. filter coefficients). The modified program is available in Appendix B "Probability of Overflow Error Calculation Software" on page 150.

MODULUS computes the PGF of each output polynomial coefficient, after performing an inner product over the filter blocklength. The probability of error is calculated by summing all the probabilities of output coefficients $c_k$ for which $|c_k| > \frac{M - 1}{2} - 1$.

Figure 3.19 Histogram of polynomial coefficients with enhanced mapping

For a filter blocklength of 53 with 10 bit input bitlength, $M=257$ and $X=8$, the probability of overflow error, for the worst case coefficient $X^2$, is 0.04%. This is below the empirical limit of 0.05% chosen based on numerous DSP simulations [111]. The error for the MRRNS system, without the enhanced mapping, is an overwhelming 40.3% for this example, making it unacceptable for the filter design. The table below shows probability
of error calculation for various mapping parameters, with and without the enhanced polynomial mapping. The results show that a single modulus, 257, is sufficient in most cases with the enhanced mapping, where the probability of errors are below the empirical limit. In the three instances where the probability of error exceeds 0.05%, adding the modulus 5 reduces the error to zero for all three cases. However, this is not the case where the enhance mapping has not been used. Without the enhance mapping, a composite modulus $257 \times 17$ is required for designs to maintain an acceptable probability of error.

Table 3.5 Probability of error calculations

<table>
<thead>
<tr>
<th></th>
<th>with enhanced mapping</th>
<th>without enhanced mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>prob. of error</td>
<td>prob. of error</td>
</tr>
<tr>
<td></td>
<td>(M=257)</td>
<td>(M=257 x 5)</td>
</tr>
<tr>
<td>$X=2$, bitlength=10, N=200</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>$X=4$, bitlength=11, N=120</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>$X=4$, bitlength=15, N=120</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>$X=8$, bitlength=10, N=160</td>
<td>1.25%</td>
<td>0%</td>
</tr>
<tr>
<td>$X=8$, bitlength=10, N=53</td>
<td>0.04%</td>
<td>0%</td>
</tr>
<tr>
<td>$X=8$, bitlength=13, N=160</td>
<td>11.72%</td>
<td>0%</td>
</tr>
<tr>
<td>$X=8$, bitlength=16, N=40</td>
<td>0.006%</td>
<td>0%</td>
</tr>
<tr>
<td>$X=8$, bitlength=19, N=40</td>
<td>4.46%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>with enhanced mapping</td>
<td>without enhanced mapping</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td></td>
<td>prob. of error (M=257)</td>
<td>prob. of error (M=257)</td>
</tr>
<tr>
<td></td>
<td>(M=257 x 5)</td>
<td>(M=257 x 17)</td>
</tr>
<tr>
<td>$X=16$,</td>
<td>0.009%</td>
<td>17.31%</td>
</tr>
<tr>
<td>bitlength=13,</td>
<td>0%</td>
<td>1.05%</td>
</tr>
<tr>
<td>$N=20$</td>
<td></td>
<td>0.027%</td>
</tr>
</tbody>
</table>

### 3.7 Summary

This chapter has provided an in-depth look at MRRNS, and its polynomial mapping. We have discussed the different forms of polynomial representation, based on the number of indeterminates, and also the different mappings based on the input binary representation. We have also examined the effect of the different polynomial representations on the channel overflow. From these discussions a new mapping strategy has been introduced that makes use of the redundant representation of the input data in signed digit form. We have provided both theory and experimental results that prove the efficacy of this new mapping scheme in drastically reducing the overflow error in single prime modulus MRRNS implementations.
Chapter 4

Architecture

4.1 Introduction

An important block in the implementation of DSP algorithms is the inner product processor. Previous work on the MRRNS system resulted in the design of a finite inner product step processor (referred to as the Fermat ALU) which operated over the modular ring $257 \times 17$. The architecture of the ALU was based on the fact that the moduli were Fermat primes (primes of the form $2^{2^k} + 1$). Using Fermat primes allows the inner product processor to operate over the half-index domain using index calculus for the multiplication and Leibowitz's diminished-ones representation [61] for the accumulation. This was found to be a better solution than the Zech Logarithm [4] approach recently adopted by Zelniker and Taylor [118] for their Gauss machine [67]. The original Fermat ALU (mod 257) will be presented in this chapter, along with the improved version of the ALU. Also a brief description of index calculus and diminished one's arithmetic will be included for completeness.
4.2 Index Calculus Residue System

If we use a Residue Number System that has a moduli set where all of the moduli are prime numbers, then we can use a property of Galois fields to map multiplication to addition. It is well known that for any prime modulus \( p \) there exists some \( g \in \mathbb{Z}_p \) that generates all non-zero elements of \( GF(p) = \{ \mathbb{S} : \oplus_p, \otimes_p \} \). That is, any non-zero element of \( \mathbb{Z}_p \) can be represented by \( g^k \) where \( k \in \{0, 1, 2, ..., p-2\} \). Since we can represent all elements of \( GF(p) - \{0\} \) as exponents, we can perform multiplication via addition. The cyclic nature of residue techniques allows multiplication to be performed via index addition, but this is at the cost of an increase in the complexity of addition. Normally, we would define a complete logarithmic system in which addition is also defined, and the early papers on Galois field implementation of filters define a complete index calculus system this way (e.g., [118]). In the case of simple inner product computations, however, where the final result is required to be mapped from the logarithmic domain, we may perform this mapping prior to the accumulation operator, thus removing the need to define addition over the logarithmic mapping. Therefore, after multiplication, the products are converted from the logarithmic representation to the ordinary representation of the finite field and then accumulated (e.g., [48]). This is the technique we have adopted for our final implementation.

For completeness we will first introduce the techniques for implementing index calculus in a completely defined system. We will then discuss the product inverse mapping technique and introduce the enhanced Fermat ALU.

Index Calculus Multiplication

From Theorem 2.7. on page 15, we observe that for all \( a, b \in GF(p) - \{0\} \):

\[
\begin{align*}
    a &= g^a \mod p \\
    b &= g^b \mod p
\end{align*}
\]  

(4.1)
where \( g \in GF(p) \) generates all the nonzero elements of \( GF(p) \). We will identify the forward mapping as, for example, \( \alpha = \mathbb{J}_g(a) \). Eqn. (4.2) is a mapping between the additive group, \( G_\oplus = \{(\{S\} - (p - 1)) : \oplus_{p-1}\} \), and the multiplicative group, \( G_\otimes = \{(\{S\} - (0)) : \otimes_p\} \), where \( g \) is a generator of \( G_\otimes \); i.e. \( g \mid_p \forall i < p - 1 \) generates all elements of \( G_\otimes \)

\[
a \otimes_p b = g^{(\alpha \otimes_{p-1}, \beta)}
\]

(4.2)

The product of \( a \) and \( b \) can be calculated according to Eqn. (4.2), recalling that the order of \( GF(p) \) is \( p-1 \).

Eqn. (4.2) suggests that in order to perform multiplication \( a \otimes_p b \), first the values of \( \alpha \) and \( \beta \) are looked up using forward tables, then they are added modulo \( (p-1) \), and finally the result corresponding to the indexed sum is looked up in an inverse table. Additional circuitry must be included for the case of either or both operands being zero. This can be implemented by introducing a tag, \( \text{NAN} \) (not a number), which sets the product of the numbers to zero. The representation of an element of \( GF(p) \) is thus given by an \( n \)-bit field for the index and a one-bit tag for \( \text{NAN} \). \( \mathbb{J}_x(\text{NAN}) \) is represented by \( n \) zeros and the \( \text{NAN} \) tag set to one.

**Index Calculus Addition**

To perform addition, we can simply factor out the addend and store all combinations of the remaining factor in a ROM. In essence we invoke the relationship of Eqn. (4.3) (defined over \( GF(p) \)).

\[
a \oplus_p b = b \oplus_p ([a \otimes_p (b)^{-1}] \oplus_p 1)
\]

(4.3)
We will refer to this as the *slide-rule trick* [48] since this was a popular technique for performing general addition on a slide-rule which uses logarithmic scales. We have to allow for the possibility of invalid mappings, such as either input being zero, but this is easily implemented using quite simple logic.

If we consider the inner product of Eqn. (4.4):

\[
c_{out} = \left( a_{in} \oplus_p b_{in} \right) \oplus_p c_{in}
\]

\[
a_{out} = a_{in}
\]

then we can use index calculus, as in Eqn. (4.5)

\[
\gamma_{out} = \gamma_{in} \oplus_p \psi(\alpha_{in} \oplus_p b_{in} \oplus_p -1 \gamma_{in} 1)
\]

where \( \psi(x) = \mathcal{F}_g(x^r \oplus_p 1) \). A ROM stores the forward mapping function, \( \psi \), and a mod \( p-1 \) adder/subtractor is used to form the \( n \)-bit address input. Note that using this technique we have converted a general binary (two-variable) input table lookup to a unary (single-variable input) lookup.

Addition over general moduli has been explored by several authors (a compendium can be found in [93]), more recent work from University of Windsor can be found in [13].

### 4.3 Diminished -1 Addition

We now discuss the product-mapping technique used with our enhanced Fermat ALU. We note that the index calculus multiplier uses binary adders (addition modulo \( 2^B \)) to compute the product index. We may also use binary adders to compute the modulo \( 2^B + 1 \) addition by invoking a special representation for the Fermat field elements, and we use a coding method termed *diminished-1 (D1) representation*, to represent these elements. This coding scheme was first introduced by Leibowitz [61], and gives an exact representation
of the numbers in this system. It is an improvement over the coding scheme proposed by Burrus [2] which introduces an input quantization error, due to the fact that a \(2^n\)-bit arithmetic is used to represent, the \(2^n+1\)-bit system\(^1\).

In order to represent all the integers in the ring of integers modulo \(F_r\), (where the prime modulus is \(p = 2^{2^l} + 1 \) \(2^l + 1\) bits are needed. In order to overcome the need to use this additional bit in performing the addition operation, the representation of the element is modified, such that the additional bit is one when the element being represented is zero, and otherwise is equal to zero. This is achieved by subtracting one from the binary representation of the integers in the ring. Therefore, the diminished-l representation of a number, \(D(A)\), will be \(A - 1\).

Before describing the multiplier, the following definitions are introduced.

**Definition 4.1** Let \(a\) be an integer. Then the diminished-l representation of \(a\) is

\[
D(a) = (a - 1) \mod (2^{2^l} + 1).
\]

**Definition 4.2** Let \(a\) and \(b\) be integers. Then the diminished-l operations are defined as:

\[
D(a) \oplus D(b) = D(a + b) \\
D(a) \ast D(b) = D(a \cdot b)
\]

(4.6)

where \(\oplus\) and \(\ast\) denote the symbol for diminished-l addition and multiplication. Using these definitions, the relationship between binary addition and diminished-l addition can be described by the following lemma.

**Lemma 1:** Let + and \(\oplus\) denote the symbols for binary and diminished-l addition, respectively. For the integers \(a\) and \(b\):

---

1. The extra bit is used in the representation of \(2^{2^l} = -1 \mod F_l\)
\[ D(a) \oplus D(b) = D(a) + D(b) + 1 \mod (2^n + 1) \quad (4.7) \]

Using the \( Dl \) representation we are able to perform modulo \( p=2^n \) addition using a \( 2^n \)-bit binary adder (the same size binary adder used for implementing index calculus). In the \( Dl \) representation the digits \( 1 \ldots p - 1 \) have the binary representation \( 0 \ldots p - 2 \) (which are uniquely defined using \( 2^l \) binary digits), and \( 0 \) is represented by \( p-1 \) which requires \( 2^l + 1 \) binary digits. We demonstrate this as follows:

Normally we compute modulo \( p \) addition as shown in Eqn. (4.8), where the two cases correspond to the state of the carry out of the modulo \( p \) adder.

\[ a \oplus_p b = \begin{cases} 
  a + b & a + b < p \\
  a + b - p & a + b \geq p 
\end{cases} \quad (4.8) \]

Using a \( Dl \) representation for a non-zero \( b \) variable, consider addition performed modulo \( p-1 \), as shown in Eqn. (4.9). The two cases are based on the state of the carry out of the modulo \( p-1 \) adder, and correspond to the same cases as in Eqn. (4.8).

\[ a \oplus_{p-1} Dl(b) = \begin{cases} 
  Dl(a \oplus_p b) & a + (b - 1) < p - 1 \\
  a \oplus_p b & a + (b - 1) \geq p - 1 
\end{cases} \quad (4.9) \]

By performing a further modulo \( p-1 \) addition with the input carry set to the complement of the carry out value, both cases will produce the correct binary representation of the modulo \( p \) addition. Since modulo addition is associative, it is not necessary to correct each addition in a chain of additions. We simply carry any addition correction to the next addition in the chain; this allows a complete feedforward implementation of the addition correction with very little overhead (an inverter on the output carry and a final correcting addition). To clarify the procedure consider the addition chain computation in Table 4.1 where \( p=17 \). Row 1 is the input to the running accumulator, Row 2 is the correct result (accumulation modulo 17) Row 3 is the output of the \( Dl \) adder and Row 4 the carry out from the \( Dl \) adder. Since the \( Dl \) adder inverts the carry in, we start the chain with the
input carry set to '1'. Note that the correct result appears when the carry out is '1' and we obtain the $D1$ representation when the carry out is '0'.

<table>
<thead>
<tr>
<th>Table 4.1 Mod 17 Chain Addition Using D1 Representations and Mod 16 Adders</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
</tr>
<tr>
<td>Mod 17 sum</td>
</tr>
<tr>
<td>D1 Adder</td>
</tr>
<tr>
<td>Carry Out</td>
</tr>
</tbody>
</table>

The importance of using the $D1$ approach is that it only requires two binary adders, versus three for the slide-rule trick using index calculus.

4.4 Original Fermat ALU

The original work on the Fermat ALU design was performed by W. Luo and G.A. Jullien [48], who introduced two variations for the ALU architecture. Both variations were designed for Fermat prime moduli. The moduli chosen were 257 and 17. The reason for this choice was that 257 alone would result in large replication factors for useful dynamic ranges in the MRRNS system. and use of the 5th Fermat number which is also prime.

$2^{16} + 1 = 65537$, would be too large for the required look-up table mapping operations required between the multiplicative and additive sub-groups. The embedding of a 257 x 17 RNS system, however, provided ample range but with an attendant overhead of at least 25% in area and power compared to a single 257 modulus ALU. The other smaller Fermat primes (3, 5), did not seem to offer any advantages, either in terms of augmenting the embedded RNS or in replacing 17 in the two modulus RNS.

The range of 257 x 17 is an insufficient output dynamic range for practical use. However, using the MRRNS technique this range can be expanded through the replication channels to accommodate useful output dynamic ranges.
The basic function of the ALU is to implement two independent MACs over the fields GF(17) and GF(257).

\[
C' = (A \oplus_{257 \times 17} B) \oplus_{257 \times 17} C.
\] (4.10)

Two different architectures for the Fermat Number MAC were explored. The first technique utilizes the *slide-rule trick* to maintain all data in the index domain; i.e., we apply Eqn. (4.4) for the multiplier and accumulator. The second technique uses a diminished-ones (DL) representation technique [61] to perform addition over each Galois field, and uses index calculus to implement multiplication. A study of the two architectures revealed that the diminished-ones representation provided a lower hardware solution (more architectural details can be found in [48]). This second architecture will be discussed in the following section. For brevity, only the Mod 257 MAC is discussed.

**MAC Implementation in the half-index domain**

This design features an index-domain multiplication \((A \oplus B)\) and a normal domain addition \((\oplus C)\). The indices of \(A, B\) are added in an 8-bit binary adder, and the sum is passed to a ROM of size \(256 \times 8\). The ROM stores the operation

\[
l^{-1}(\alpha \oplus \beta) \oplus (-1) = A \oplus B \oplus (-1).
\]

The structure of the MAC is shown in Figure 4.1. The addition is modulo 257 (over the Galois Field), using a diminished-ones adder [61]; the output of the ROM is \(A \oplus B \oplus (-1)\). The other input, \(C\), and the output, \(C'\), are also in the diminished-one form. Based on the diminished-ones code, the carry from the MSB is inverted and added to the LSB. In this design, the iterative MACs are pipelined, and the addition is passed to the next MAC as the carry-in.
4.5 A New Half index Domain MAC

A new design for the half index domain MAC is proposed here. This design modifies the ROM size and also adds some extra logic to consider the case where the input and output accumulated value is zero and needs to be handled as a special case in diminished one addition.

4.5.1 ROM size reduction

The work in [80] presented the half index domain MAC implementation in an FPGA. Because the FPGA is limited in terms of resources for implementing the full lookup table, a method for reducing the tables size to a quarter of the original size was proposed. By adding some extra logic the full table can be derived from the reduced table.
To reduce the size of this look up table, the following properties of finite rings were used:

<table>
<thead>
<tr>
<th></th>
<th>case 1</th>
<th>case 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{p-1}{4} \leq X &lt; \frac{p-1}{2}$</td>
<td>$g^x = (-j) \otimes_p g^{\frac{x - \frac{p-1}{4}}}{4}$</td>
<td>$g^x = j \otimes_p g^{\frac{x - \frac{p-1}{4}}}{4}$</td>
</tr>
<tr>
<td>$\frac{p-1}{2} \leq X &lt; \frac{3(p-1)}{4}$</td>
<td>$g^x = -g^{\frac{x - \frac{p-1}{2}}}{2}$</td>
<td>$g^x = -g^{\frac{x - \frac{p-1}{2}}}{2}$</td>
</tr>
<tr>
<td>$\frac{3(p-1)}{4} \leq X &lt; p-1$</td>
<td>$g^x = j \otimes_p g^{\frac{x - \frac{3(p-1)}{4}}}{4}$</td>
<td>$g^x = -j \otimes_p g^{\frac{x - \frac{3(p-1)}{4}}}{4}$</td>
</tr>
</tbody>
</table>

where $\pm j$ is a root of $\left(Z^2 \otimes_p 1\right)$. Over GF(257), $j$ is equal to 16, which makes the procedures shown Table 4.2, particularly easy for hardware implementation. Generators, $g$, for case 1 and case 2 over GF(257), are listed in Table 4.3. The method proposed in [80] uses the 6 LSB of the address to the full ROM to look up the value in the reduced ROM, then by examining the seventh and eighth bits, determines whether the output of the reduced ROM is the desired value or if operations such as shift right/left or negation need to be performed to recover the desired value. The hardware requirements for this method are multiplexers, bidirectional shift register and inverters. Also, care must be taken in the selection of the generator so as to guarantee that none of the values stored in the full ROM are multiples of 16. This restricts the choice of the generator to the shaded cells in Table 4.3.

A simpler method is to reduce the ROM by half. The 7 LSB bits of the address to the full ROM are used to look up a value in the reduced ROM. By examining the MSB bit we can determine if the output from the ROM is the desired value or if the value needs to be negated. Hence the hardware reduces to a small multiplexer and inverters. This method places no restriction on the choice of the generator.
Table 4.3 All the primitive roots for M=257

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>6</th>
<th>7</th>
<th>12</th>
<th>14</th>
<th>19</th>
<th>24</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>38</td>
<td>45</td>
<td>47</td>
<td>48</td>
<td>51</td>
<td>53</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>66</td>
<td>69</td>
<td>76</td>
<td>77</td>
<td>90</td>
<td>94</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>103</td>
<td>105</td>
<td>106</td>
<td>112</td>
<td>119</td>
<td>125</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>132</td>
<td>138</td>
<td>145</td>
<td>151</td>
<td>152</td>
<td>154</td>
<td>155</td>
<td></td>
</tr>
<tr>
<td>161</td>
<td>163</td>
<td>167</td>
<td>180</td>
<td>181</td>
<td>188</td>
<td>191</td>
<td>192</td>
<td></td>
</tr>
<tr>
<td>201</td>
<td>204</td>
<td>206</td>
<td>209</td>
<td>210</td>
<td>212</td>
<td>219</td>
<td>224</td>
<td></td>
</tr>
<tr>
<td>229</td>
<td>233</td>
<td>238</td>
<td>243</td>
<td>245</td>
<td>250</td>
<td>251</td>
<td>255</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>10</td>
<td>20</td>
<td>27</td>
<td>37</td>
<td>39</td>
<td>40</td>
<td>41</td>
</tr>
<tr>
<td>43</td>
<td>54</td>
<td>55</td>
<td>63</td>
<td>71</td>
<td>74</td>
<td>75</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>82</td>
<td>83</td>
<td>85</td>
<td>86</td>
<td>87</td>
<td>91</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>97</td>
<td>101</td>
<td>107</td>
<td>108</td>
<td>109</td>
<td>110</td>
<td>115</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>131</td>
<td>142</td>
<td>147</td>
<td>148</td>
<td>149</td>
<td>150</td>
<td>156</td>
<td>160</td>
<td></td>
</tr>
<tr>
<td>164</td>
<td>166</td>
<td>170</td>
<td>171</td>
<td>172</td>
<td>174</td>
<td>175</td>
<td>177</td>
<td></td>
</tr>
<tr>
<td>179</td>
<td>182</td>
<td>183</td>
<td>186</td>
<td>194</td>
<td>202</td>
<td>203</td>
<td>214</td>
<td></td>
</tr>
<tr>
<td>216</td>
<td>217</td>
<td>218</td>
<td>220</td>
<td>230</td>
<td>237</td>
<td>247</td>
<td>252</td>
<td></td>
</tr>
</tbody>
</table>

An algorithm based on Table 4.2 over GF(257) is suggested as follows:

1. $X = \alpha \oplus_{256} \beta, \bar{X} \equiv \begin{bmatrix} x_7, x_6, \ldots, x_0 \end{bmatrix}$

2. Look-up $g^{-x}$ for $\bar{X} \equiv \begin{bmatrix} x_6, x_5, \ldots, x_0 \end{bmatrix}$; i.e. $0 \leq \bar{X} \leq 127$.

3. If $x_7=1$, negate the result of the previous step: i.e. compute $1-X$.

4.5.2 Modification for Diminished-1 Accumulation

As discussed earlier (Section 4.3), zero is treated as a special case in diminished-1 representation, and provision must be made for the case where the incoming accumulated
value $C-l$ and the outgoing accumulated value $C'-l$ are zero. A flag $CN_{AN}$ is used to identify a zero value and simple logic circuitry examine this flag to determine whether to pass the incoming accumulation value or the number 10000000 (zero in the diminished-1 representation) to the diminished-1 adder. Also additional logic needs to be added to the output of the adder to determine from $C'-l$ and $Carry(0)$ whether the output is zero, in which case an output $NAN$ flag will be set to be passed on to the next MAC.

A complete blockdiagram of the modified MAC is shown in Figure 4.2.

![New MAC block diagram](image)

**Figure 4.2 New MAC block diagram**

4.6 Summary

This chapter presents an architecture for a finite inner product processor (Fermat ALU) to be used with the MRRNS system. The efficiency in the processor design arises from the fact that the operations are performed over Fermat primes. This allows for multiplication to be performed with index calculus and addition to be performed using a diminished-1 representation, effectively reducing the hardware to binary adders and look-up tables. A
brief discussion of index calculus multiplication and diminished-1 addition has also been presented in this chapter, along with previous architectural designs of the ALU. Finally modifications and enhancements to the original Fermat ALU are discussed, resulting in reduced hardware.
Chapter 5

A MRRNS FIR Array
Case Study

5.1 Introduction

In the previous chapters we have provided the resources for designing an efficient DSP architecture that implements inner product type computations. This system uses the MRRNS along with an enhanced input mapping to represent the data as polynomials. The modified Fermat ALU is used as the main building block for this architecture. In this chapter, the various components that comprise the entire system, including the input and output mapping blocks, and processing channels for the additional polynomial coefficients, will be developed. We finish the chapter by first generating a floorplan for the general design of a MRRNS FIR array and then simulating a complete 53-tap filter, targeted for use in a video interpolation application.

5.2 Input Mapping

5.2.1 Polynomial mapper

In order to reduce the polynomial coefficient range we perform the conversion defined in Eqn. (3.33). Figure 5.1 shows a block diagram of the technique used for this enhanced polynomial...
mapping; the memory blocks are $2^{3+1} \times \beta$ ROMs. The input data stream is first simply wire mapped to polynomial coefficients $c_0, \ldots, c_{n-1}$, by segmenting the input bitlength into $\beta$ bit segments. These coefficients are then passed through the mapper to produce the new polynomial coefficients $c_0', \ldots, c_n'$. For a heavily pipelined system (i.e. where the table is registered on both the input and output) we can use a switching tree approach to implement the ROMs [47].

**Figure 5.1 Block diagram of the input polynomial mapper**

The small lookup tables for the input polynomial mapper are designed using minimized switching trees [47]; these will be further discussed in Section 5.7.1. The inputs to the mapper are in signed binary representation and the output of the mapper is in 2's complement representation. All but the MSB coefficient from the enhanced polynomial mapper are sent to the evaluation map stage. The MSB coefficient is sent to the binary
channel and is used in performing the polynomial addition, outside of the finite field computations (see Eqn. (3.35) on page 63).

5.2.2 Evaluation map

The evaluation map is a matrix-vector multiplication, and it can be computed using an array of MACs. This map multiplies the vector of polynomial coefficients with the Vandermonde matrix of the ideal (Eqn. (3.9) on page 44). The roots of the ideal are 0 and ±i where \( i = \{1, \ldots, \lceil (B-1)/\beta \rceil - 1 \} \). This matrix operation is shown in Eqn. (5.1), where

\[
\begin{bmatrix}
1 & 0 & \ldots & (r_0)^n & (r_0)^{n+1} & \ldots & (r_0)^{2n} \\
1 & 1 & \ldots & (r_1)^n & (r_1)^{n+1} & \ldots & (r_1)^{2n} \\
1 & -1 & \ldots & (r_{-1})^n & (r_{-1})^{n+1} & \ldots & (r_{-1})^{2n} \\
\vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
1 & r_{(\beta+1)/2} \cdot (r_{-1})^{n+1} \cdot (r_{n+1})^{n+1} & \ldots & (r_{(\beta+1)/2})^n & (r_{(\beta+1)/2})^{n+1} & \ldots & (r_{(\beta+1)/2})^{2n} \\
1 & (r_{-1})^{n+1} \cdot (r_{n+1})^{n+1} & \ldots & (r_{(\beta+1)/2} \cdot (r_{-1})^{n+1} \cdot (r_{n+1})^{n+1})^n & (r_{(\beta+1)/2} \cdot (r_{-1})^{n+1} \cdot (r_{n+1})^{n+1})^{n+1} & \ldots & (r_{(\beta+1)/2} \cdot (r_{-1})^{n+1} \cdot (r_{n+1})^{n+1})^{2n} \\
\vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
1 & r_n & \ldots & (r_n)^n & (r_n)^{n+1} & \ldots & (r_n)^{2n} \\
1 & r_{-n} & \ldots & (r_{-n})^n & (r_{-n})^{n+1} & \ldots & (r_{-n})^{2n}
\end{bmatrix}
\begin{bmatrix}
c_0^r \\
c_1^r \\
\vdots \\
c_n^r \\
0 \\
0 \\
\end{bmatrix}
\]

(5.1)

Since the input polynomials have the \( n \) highest power coefficients set to zero (the Vandermonde matrix is \((2n+1)\) by \((2n+1)\)), the \( n \) rightmost columns in the forward map matrix will not be used. Hence the matrix operation is reduced to:
\[
\begin{bmatrix}
1 & 0 & \cdots & 0 \\
1 & 1 & \cdots & 1 \\
1 & -1 & \cdots & (-1)^n \\
\vdots & \vdots & \ddots & \vdots \\
1 & r_{(-1)^{n-1} \times \frac{n}{2}} & \cdots & \left(r_{(-1)^{n-1} \times \frac{n}{2}}\right)^n \\
1 & r_{(-1)^{n-1} \times \frac{n+1}{2}} & \cdots & \left(r_{(-1)^{n-1} \times \frac{n+1}{2}}\right)^n \\
\vdots & \vdots & \ddots & \vdots \\
1 & r_{(n-1)} & \cdots & (r_n)^n \\
1 & r_{-(n-1)} & \cdots & (r_n)^n \\
\end{bmatrix}
\]

(5.2)

If we look at the first three rows of the Vandermonde matrix where \(\{0, \pm 1\}\) are used as part of the root set for the ideal, then the first row involves no calculations and the second and third row only require additions and subtractions. For any roots, the first column of the forward map matrix only contains the element '1' which simply involves an input connection to the accumulator for the second column in the row. For the target \((2n+1)\)th order map, there are, therefore, only the \(2n-2\) final rows with \(n\) out of the \(2n+1\) columns requiring a MAC to implement the vector multiplication. This is a total of \((n) \times (2n + 1)\) Fermat ALUs for the input map.

The Fermat ALUs have three inputs, two are for the index calculus multiplication and the other for the diminished ones addition. The output from the Fermat ALU's will be in diminished ones representation; hence, all but the LSB coefficient of the polynomial coefficients from the polynomial mapper stage need to be converted to their corresponding index values. The LSB coefficient needs to be converted to a diminished ones representation as it is used as a diminished ones input to the Fermat ALU.
5.3 Computational Channels

5.3.1 Finite ring channels

The number of computation channels is equal to the degree of the ideal and from the previous section will be $2n-2$. The computational channels consist of cascaded Fermat ALUs, with the number of ALUs equal to the blocklength of the inner product, $N$. The ALUs used are based on the modified Fermat ALU (see Figure 4.2 on page 87). The inputs and outputs of the Fermat ALU are shown block diagram in Figure 5.3.

The ROM used in this ALU is the reduced ROM discussed in Section 4.5.1 on page 84. The reduction uses the bit-reversal property of the upper and lower table contents, in that the first half of the lookup table is the bit inverse of the second half for selected index generators. This effectively means that only half the look up table needs to be stored in the
ROM, with the inverse or true values being selected by the most significant bit, as shown in Figure 5.4.

Figure 5.3 Block Diagram of Fermat ALU

Fermat ALU

Index

α(in)

β(in)

α_nav(in)

β_nav(in)

\{ C \rightarrow 1 \}

D(1)

\{ Carry(in) \}

\{ ζ Nav(in) \}

CK

MUX

MUX

LUT

\rightarrow α(out)

\rightarrow β(out)

\rightarrow α_{NAV}(out)

\rightarrow β_{NAV}(out)

\rightarrow ζ_{NAV}(out)

\rightarrow Carry(out)

\rightarrow CK

\rightarrow CK_{coeff}

\bullet latch

Figure 5.4 Minimized ROM

The diminished ones outputs from the evaluation stage need to be converted to their corresponding index values to be used as inputs to the ALUs. This is done using a look-up table which also looks up values for \( \alpha_{NAV} \) and \( \beta_{NAV} \). The diminished one input to the
first ALU in the cascade chain is equal to zero (in diminished ones representation) with $c_{NAV}$ equal to one and $C-1$ equal to 100000000 (binary).

### 5.3.2 Binary Channel

The FIR array implements the inner product of two polynomials: the polynomials are represented using the enhanced map. The polynomial multiplication is simplified, based on the knowledge that the enhanced mapping will produce polynomials whose MS coefficient is $\in \{1, 0, 1\}$. This simplification is shown in Eqn. (3.35) on page 63. The polynomial multiplication in Eqn. (3.35), is performed using the finite field computational channels. The remaining polynomial additions are performed in a separate binary channel.

The additions are performed concurrently with the stages of the finite field computational channels, therefore there are $N$ stages of additions. The additions are controlled by the MS coefficients of the two input streams. The possible additions in each stage are summarized in the table below.

<table>
<thead>
<tr>
<th>Values of $ca'<em>{n+1}$ and $cb'</em>{n+1}$</th>
<th>Addition result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ca'<em>{n+1} = 0$ and $cb'</em>{n+1} = 0$</td>
<td>0</td>
</tr>
<tr>
<td>$ca'<em>{n+1} = 0$ and $cb'</em>{n+1} = 1$</td>
<td>$X^{(n+1)} \cdot A'(X)$</td>
</tr>
<tr>
<td>$ca'<em>{n+1} = 0$ and $cb'</em>{n+1} = -1$</td>
<td>$-(X^{(n+1)} \cdot A'(X))$</td>
</tr>
<tr>
<td>$ca'<em>{n+1} = 1$ and $cb'</em>{n+1} = 0$</td>
<td>$X^{(n+1)} \cdot B'(X)$</td>
</tr>
<tr>
<td>$ca'<em>{n+1} = 1$ and $cb'</em>{n+1} = 1$</td>
<td>$X^{2(n+1)} \cdot X^{(n+1)} \cdot B'(X) + X^{(n+1)} \cdot A'(X)$</td>
</tr>
<tr>
<td>$ca'<em>{n+1} = 1$ and $cb'</em>{n+1} = -1$</td>
<td>$-X^{2(n+1)} \cdot B'(X) + X^{(n+1)} \cdot A'(X)$</td>
</tr>
<tr>
<td>$ca'<em>{n+1} = -1$ and $cb'</em>{n+1} = 0$</td>
<td>$-X^{(n+1)} \cdot B'(X)$</td>
</tr>
<tr>
<td>$ca'<em>{n+1} = -1$ and $cb'</em>{n+1} = 1$</td>
<td>$-X^{2(n+1)} \cdot X^{(n+1)} \cdot B'(X) + X^{(n+1)} \cdot A'(X)$</td>
</tr>
<tr>
<td>$ca'<em>{n+1} = -1$ and $cb'</em>{n+1} = -1$</td>
<td>$X^{2(n+1)} - X^{(n+1)} \cdot B'(X) - X^{(n+1)} \cdot A'(X)$</td>
</tr>
</tbody>
</table>
The result of the additions from each stage are then summed with the next stage. The final accumulated result from the binary channel is then sent to the final adder stage to be summed with the output polynomial from the finite field computational channel (after output mapping). Tri-state buffers can be used to pass the appropriate values to be summed in each stage depending on the value of $ca'_n + 1$ and $cb'_n + 1$. The final accumulated value in polynomial form will be expression (5.3):

$$
ca'_{n+1}cb'_{n+1}X^{2(n+1)} + (ca'_{n+1} \cdot b_n' + cb'_{n+1} \cdot a_n')X^{2n+1} + (ca'_{n+1} \cdot b_{n-1}' + cb'_{n+1} \cdot a_{n-1}')X^{2n} + \ldots + (ca'_{n+1} \cdot b_0' + cb'_{n+1} \cdot a_0')X^{n+1}
$$

5.4 Output mapper

The output mapping stage is also a matrix-vector multiplication. The matrix is the inverse of the Vandermonde matrix in Eqn. (5.1). The vector is the formed from the computational channels. The multiplication is performed using MACs, and a total of $(2n-2)2$ MACs are used. The outputs from the computational channels are in diminished one form and need to be converted to index form to be used by the MACs. The diminished one input to the MACs in the first column will be zero (in diminished one form). A block diagram of the output mapper is shown in Eqn. (5.5).

The output from the output mapper will be in diminished ones form and needs to be converted to binary before being sent to the final adder stage.
5.5 Final Adder

The inverse mapping of the output from the Fermat ALU MACs will result in a polynomial that will be summed with the values from the binary adders, as shown in expression (5.4). The polynomial is then evaluated by substituting the indeterminate value. This stage can be implemented by a CSA array.

\[
ca'_{n+1} \cdot b'_{n+1} \cdot X^{2(n+1)} + ca'_{n+1} \cdot X^{(n+1)} \cdot B'(X) + cb'_{n+1} \cdot X^{(n+1)} \cdot A'(X) + A'(X) \cdot B'(X)
\] (5.4)

Figure 5.6. shows the CSA array for a FIR array, with \(X=8\). The output dynamic range is \((2n+4)\beta+2\) bits, where \(n\) is the degree of the input polynomial.
5.6 FIR Array Floorplan

The floor plan for implementing an FIR array is shown in Figure 5.7. The structure is basically divided into three parts:

1. MRRNS encoding (Forward polynomial mapping):
2. MRRNS computational paths:
3. MRRNS decoding, which is divided into two steps:
   3.1. Inverse polynomial mapping:
   3.2. final addition reducing the polynomial to a binary number.

Comparing this floorplan with the floorplan suggested in [48] and shown in Figure 5.8, we see that since only modulus 257 is used, the need for mixed radix conversion is eliminated, so are input mappings/MACs for modulus 17. Even with the addition of the binary channel and the enhanced mapper, this new floorplan is more efficient in area. Also since the number of MACs used is reduced, due to the elimination of the mod17 channels and also because of the reduction in the ROM size in the ALU, this floorplan will be more power efficient.
5.7 Example of a 53 TAP filter Design

Having described the floorplan for an enhanced Fermat ALU FIR array in the previous sections, we will now provide a practical illustration by designing a specific FIR array. This array design will then be compared with the previous Fermat ALU design to obtain a measure of the efficiencies realized using this new architecture.
The FIR array we have chosen is a 53 TAP filter with $B=10$ bit data and coefficient stream (including sign bit) with polynomial indeterminate $X=8$. The array is modelled and simulated using the VerilogXL hardware description language. Based on the results from Table 3.5 on page 74, the single modulus, 257, provides sufficient polynomial coefficient dynamic range for the computations involved. The design is modelled as a fully pipelined structure, with two clocks, one for the serial loading of the filter coefficients and the other as the operational clock signal. The aim is to present a design that can be implemented as further work, which minimizes the number of input and output signals: a typical VLSI implementation will have pin limitations.

### 5.7.1 Input Mapping Stage

For this example, the number of input polynomial coefficients before the enhanced mapping will be $9/3=3$ (since $X=2^3$), and the bitlength of the input, excluding the sign, is 9 bits. The range of the input coefficients at this point is $[-7, 7]$. Let us represent the input polynomial using the simple map as in expression (5.5):

$$a_2X^2 + a_1X^1 + a_0X^0$$

(5.5)

We assume that $X = 8$ and so the enhanced mapping restricts coefficient values to the range $[-3, 3]$. For values of $|a_i| < 4$, we need do nothing; for values of $|a_i| > 4$, we can reduce the magnitude of $a_i$ by incrementing the next highest coefficient as follows:

$$a_i = \begin{cases} X - a_i & (a_i > 4) \\ X + a_i & (a_i < -4) \end{cases}$$

(5.6)

$$a_{i+1} = \begin{cases} a_{i+1} + 1 & (a_i < -4) \\ a_{i+1} - 1 & (a_i > 4) \end{cases}$$
where we assume that the indeterminate, \( X \), is replaced by 8. The conversion process is performed in sequence, starting from the least significant coefficient; the ROMs store the function shown in (5.7), with the enhanced mapper block diagram shown in Figure 5.9.

\[
\begin{align*}
\text{for } (|a_i| < 4) & \quad a'_i = a_i \quad C_i = 0 \\
\text{for } (|a_i| > 4) & \quad a'_i = \pm X + a_i \quad C_i = \pm 1
\end{align*}
\] (5.7)

**Figure 5.9 Enhanced polynomial mapper**

---

**Minimized mapping ROM**

Rather than implement a standard row/column ROM, we have elected to use a circuit technique from the VLSI Research Group at Windsor [47], referred to as a Switching Tree ROM. The ROM is built as an \( n \)-level multiple output binary tree of transistors with the bottom row “programmed” with the ROM contents (transistors removed for ‘1’ output and retained for a ‘0’ output). The resulting graph (transistor array) is minimized based on simple rules [46]. The 3-output bit minimized switching tree for the coefficient mapper is shown in Figure 5.10: we have not included the True-Single-Phase Clocked (TSPC) dynamic latches [1] that are normally part of the switching tree structure, to simplify the schematic. The latches are included as leaf cells on the layout block shown in Figure 5.11.
Mapping ROM delay simulation

In order to use the switching tree ROM in our VerilogXL simulation, we require to measure the delay of the circuit and insert this into the behavioural code used to describe the characteristics of the mapper. In Figure 5.12 we see the SPICE results for the worst-case delay of the tree (exciting a path which has the greatest number of series transistors): this delay is 0.29ns. Adding on to this delay is the TSPC latch delay of 0.6ns
(as shown in the SPICE plot of Figure 5.13). We therefore have used a delay of 0.9ns in the behavioural code for this input mapper.

**Figure 5.12** Spice results from switching tree

**Figure 5.13** TSPC latch SPICE results
5.7.2 Evaluation Map

As the input originally is of degree 2, we will require an ideal of at least degree 5 to prevent polynomial order overflow following a polynomial multiplication. For this example the following ideal is chosen:

\[
g(X) = X(X - 1)(X + 1)(X - 2)(X + 2)
\]

where the roots of the ideal are \(0, \pm 1, \pm 2\). The Vandermonde matrix for the evaluation map will then be:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 \\
1 & -1 & 1 & -1 & 1 \\
1 & 2 & 4 & 8 & 16 \\
1 & -2 & 4 & -8 & 16 \\
\end{bmatrix}
\begin{bmatrix}
c_0' \\
c_1' \\
c_2' \\
0 \\
0 \\
\end{bmatrix}
\]

which can be reduced to:

\[
\begin{bmatrix}
1 & 0 & 0 \\
1 & 1 & 1 \\
1 & -1 & 1 \\
1 & 2 & 4 \\
1 & 2 & -4 \\
\end{bmatrix}
\begin{bmatrix}
c_0' \\
c_1' \\
c_2' \\
0 \\
0 \\
\end{bmatrix}
\]

A block diagram of the evaluation map is shown in Figure 5.14, where \(c_1', c_2'\) are converted to their index representation using a lookup table with the generator=3. \(c_0'\) is converted to a diminished ones representation since it is the accumulated value being passed to the Fermat ALUs for implementing rows 4 and 5 in (5.10).
Figure 5.14 Evaluation Map

An additional adder stage is required for rows 4 and 5, as we may have overflow from the diminished ones output from the Fermat ALU, i.e., \( \text{Carry(out)} = 1 \). (See the block diagram of the Fermat ALU in Figure 5.3). To obtain the correct diminished ones representation, \( \text{Carry(out)} \) is inverted and summed with \( C' - 1 \).

5.7.3 Computational Channel

Binary Computational Channel

In the binary computational channel, 53 cascaded adder stages, as described in Section 5.3.2, are needed. If we let the original data representation be

\[
A(X) = a_2X^2 + a_1X^1 + a_0X^0
\]

for the data input and

\[
B(X) = b_2X^2 + b_1X^1 + b_0X^0
\]

for the filter coefficient, then we can rewrite the summation in Eqn. (5.3) as:

\[
a_3b_3 \cdot X^6 + a_3X^3 \cdot B(X) + b_3X^3 \cdot A(X) \tag{5.11}
\]

where \( a_3, b_3 \in \{-1, 0, 1\} \). Factoring out \( X^3 \) from (5.11), leaves us with a simple polynomial summation. This summation is repeated and accumulated for each filter coefficient. \( B(X) \) which represents the filter coefficients is inputted to the adders in a serial
fashion. Figure 5.15 shows a single adder stage. The binary computational channel is implemented by simply cascading 53 of the adder stages. The final result is then multiplied by $X^3$ (simple shift left operation) at the Final Addition Stage.

**Figure 5.15 Single adder stage**

Finite Field Computational Channel

The last term in eqn. (5.11) is a polynomial multiplication of two second degree polynomials, which is performed over the direct product ring over 5 computational channels, each consisting of 53 cascaded Fermat ALUs. The Fermat ALU blocks have been modified slightly to allow for the filter coefficients to be loaded in a serial fashion. The $\alpha$ input is the coefficient stream which is pipelined through the computational chain prior to beginning the inner product computation, using a separate clock signal. Once all the coefficients are in place, the clock controlling them is set to zero, in effect causing the values to be held in place. The $\beta$ input is the data stream which is pipelined through with a clock that starts the inner product computations.
5.7.4 Output Stage

The output stage performs an inverse map on the coefficient values arriving from the computational stage. These coefficients, which are in diminished ones representation, need to be converted into their index representation. They are then multiplied by the inverse of the Vandermonde matrix in eqn. (5.9) as shown below:

\[
(5.12)
\]

A block diagram for the implementation of the output stage is shown in Figure 5.17. As with the evaluation map, the outputs from this stage need to be converted to the correct diminished ones representation. Hence, in each row, the \textit{Curry(out)} from the ALUs of the final column is inverted and summed with \(C' - 1\), resulting in a subsequent sum and output carry. The sum is the corrected diminished ones representation. This representation then is converted to a standard binary representation before being sent to the \textit{Final Addition} stage. The conversion is performed by inverting the output carry and adding it to the sum from the correction adder [61].
Verilog simulation results of the output stage are presented in Appendix C "Verilog Code" on page 242.

### 5.7.5 Final Adder

The expression in (5.11) shows the final addition calculation and this can be implemented by a CSA array, as shown in Figure 5.18. In this diagram, \( y \) and \( c \) are computed as shown in eqn. (5.13).

\[
y = y_3 X^6 + y_2 X^5 + \ldots + y_0 X^3 \quad \text{from adder}
\]

\[
c = c_4 X^3 + c_3 X^2 + c_1 X + c_0 \quad \text{from MACs}
\]

The first 3 terms on the RHS of Eqn. (5.13) are polynomial additions, which are implemented outside of the direct product computation using 4 adders for each computational stage.

The polynomial is then reduced to an integer by substituting the indeterminate value. The output dynamic range for this illustrative example is 26 bits.
VerilogXL simulation results for the final adder can be found in Appendix C "Verilog Code" on page 242.

5.7.6 Complete FIR Array

The floorplan for the 53-tap filter is shown in Figure 5.19.
The blocks shown are described in the previous section. A VerilogXL description based on the this floor plan, along with simulation results, can be found in Appendix C.

5.8 Power Estimation

The floorplan described for the FIR array can be implemented in silicon, using the circuitry given in Appendix D "Circuitry" on page 315 which includes layout for the TSPC latch, 8 bit adders, ROMs, and multiplexers. These circuits have been simulated, fabricated and fully tested, with estimations of their power dissipation given in Table D.2 on page 324. Based on this information an estimation is made of the power dissipation of the new Fermat ALU, and also the 53-tap FIR array used as an example in this chapter.

Table 5.2 shows a comparison of the power dissipation of the original Fermat Inner Product Step Processor (IPSP) with 5 replications of the Fermat ALU, with the new Fermat IPSP and a binary MAC. The new Fermat IPSP show a 20% improvement in power compared to the original Fermat ALU, and 45% improvement compared to an equivalent binary MAC.

<table>
<thead>
<tr>
<th></th>
<th>Power(mW/100MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.5μm</td>
</tr>
<tr>
<td>FermatIPSP(257x17)</td>
<td>38.5</td>
</tr>
<tr>
<td>Fermat IPSP(257)</td>
<td>25.6</td>
</tr>
<tr>
<td>Fermat IPSP (257)</td>
<td>22.85</td>
</tr>
<tr>
<td>MAC with Booth algorithm</td>
<td>41.0</td>
</tr>
</tbody>
</table>

For the 53 TAP FIR array example, without the enhanced mapping, we would need to use the original Fermat IPSP (computing over \( \mathbb{Z}_{257 \times 17} \)): however, with the enhanced mapping, the Fermat IPSP (over \( \mathbb{Z}_{257} \)) is sufficient. Using the modified Fermat design, we achieve over 40% power savings in the finite field computational channels. Taking into consideration the additional hardware required for the enhanced polynomial mapper and
the additional binary computational channel, the total power estimation for the 53-tap FIR array will be 1.21W at 100MHz.

5.9 Summary

This chapter has presented an overall floorplan for the design of a FIR array using a single modulus, enhanced input polynomial mapping, and modified Fermat ALU design. The various blocks in the floorplan, such as the input mapping, output mapping and computational channels are detailed. The small ROM look-up tables, required in the enhanced input mapper, are presented as Switching Tree arrays, and a complete transistor level design of this component has been included for completeness.

As an illustrative example of a practical implementation of the FIR array floorplan, a 53-tap FIR array is presented with special attention given to the hardware design issues. The various blocks are described based on their Verilog XL models, and some results from the VerilogXL simulator are provided.
6.1 Conclusions

Although the classical finite arithmetic approach to designing DSP applications offers advantages in terms of VLSI design and architectural simplification, it suffers drawbacks due to the cumbersome forward and reverse mappings and asymmetrical computational channels. The mapping strategy based on the MRRNS approach, and modified in this work, offers all the advantages of the classical approach without the complex mapping overhead. The enhanced mapping, introduced in this thesis, improves on the classical MRRNS approach by limiting the magnitude of the polynomial coefficients that are used to represent the data. This allows for an area/power efficient implementation of large length inner product processors that are much superior to binary implementations.

6.2 Contributions

This thesis has presented a new mapping strategy, with associated architectures, for implementing general purpose inner product computations using enhanced Fermat ALU theory. The structure is based on a direct product finite polynomial ring mapping of a
redundant binary representation of the input data; in effect we exploit the double redundancy of the input representation and the mapped polynomial representation. By exploiting this redundancy, with attendant reductions in coefficient growth due to polynomial multiplication, we are able to considerably reduce the probability of overflow error.

The redundant property of the polynomial map is used to optimize the input data. By allowing a mix of positive and negative coefficients to represent any number, regardless of sign, we can reduce the maximum value of the coefficient by as much as half. This is sufficient to reduce the probability of overflow to acceptable levels using only single modulus computations, with considerable reduction in computational hardware. We trade-off this reduction in the coefficient range by increasing the degree of the polynomial representation. This turns out not to be a problem since the enhanced polynomial representation will be at most one degree higher than the polynomial representation derived from a sign and magnitude binary representation of the input data, and this can be handled with a small amount of additional hardware. We have demonstrated, for the case of FIR filter inner product applications, that this new approach allows us to implement reasonable filter lengths using only a Mod 257 ALU. The probability of overflow in the finite field channels is considerably reduced compared to an implementation without the enhanced mapping. This results in less hardware and less power dissipation and, due to the additional binary channel, an increase in the output dynamic range.

A floorplan of the overall FIR filter design is presented in this thesis. The feasibility of this design is based on calculating the probability of overflow error and showing that it is less than an empirically determined limit of 0.05%. We have modeled the errors that occur when coefficient computations overflow the ring modulus, by defining probability generating functions for the input polynomial data and coefficient streams. A practical example for a 53-tap FIR filter is detailed. The probability of error is computed to be 0.04%, compared to a 40% probability of error for the simple polynomial map. This example is discussed in detail, with each building block modelled and simulated using VerilogXL.
In terms of the efficacy of our new technique, we have estimated area and power costs for the 53-tap design and these fall well within the limits expected of modern video signal processing applications, as specified by our supporting industry, Genum Corp.

6.3 **Suggestions for Future Work**

In general, the use of multiple indeterminates adds to the complexity of the forward and reverse map and hence increases the number of replication channels. However, for very large input bitlengths, using a reasonably small single indeterminate (β<5) we will require high order polynomials which might be better represented with multiple indeterminates. There may even be a reduction in the degree of the ideal in each indeterminate and in spite of the additional indeterminates, the number of replication channels may be less than in the single indeterminate representation. A formal study of the trade-offs associated with multiple indeterminates would certainly be interesting.

We may also consider examining the ideal itself, to determine whether the choice of certain roots may result in further simplifications of the reverse mapping.

Adding fault detection and correction capabilities to the inner product application may be achieved by increasing the redundancy of the computations. This is accomplished by choosing an ideal that is several degrees higher than the output polynomial representation. Considering all the redundancies that already exist in this representation, i.e. the polynomial and binary redundancies, we may find that adding only a single computational channel may be sufficient to detect and correct and overflow fault in any of the computational channels.

Table D.2 on page 324 shows the power estimation for the various components of the Fermat ALU in both 0.5μm and 0.35μm target processes designed by other members of the VLSI Research Group\(^1\). Table 5.2 shows a comparison of the Fermat IPSP (5

---

1. Designs supervised by Dr. Binqiao Li. Post doctoral fellow in the VLSI Research Group
replications of the Fermat ALU) with a carefully designed binary multiply/accumulator; these results demonstrate a power savings of more than 30%. Further savings, as much as 50%, can be achieved by redesigning the Fermat ALU and reducing the ROM size.

A low cycle rate test performed on the ALU shows an output delay of the 8-bits dynamic adder of 3.65\,ns and 3.25\,ns for 0.5\,\mu m and 0.35\,\mu m designs respectively (see Figure D.11 in Appendix D). The Fermat ALU has maximum clock frequencies of 120\,MHz and 133 MHz for the 0.5\,\mu m and 0.35\,\mu m designs respectively. Currently the adder is the bottleneck in the Fermat ALU. By carefully redesigning the adder to match the ROM decoding and access times, the clock frequencies for the ALU can be increased.
REFERENCES


REFERENCES


Appendix A  
Properties of Number Systems

A.1 Properties of Number systems

Since the number systems of interest in this thesis are integer number systems, some general properties of these systems will be discussed here [96].

Range: The range of a number system is defined as the interval over which every integer can be represented by the system without having two numbers with the same representation. The decimal number system is an example of a number system with infinite range.

Uniqueness: A number representation is said to be unique if each number in the system has only one representation.

Redundancy: A number system is defined to be redundant if there are fewer numbers than there are combinations of digits. Therefore, for some combinations of the digits, a defined number may not exist. Alternatively different combinations may correspond to the same number. Nonuniqueness obviously implies redundancy.
Weighted Number System: A number system is said to be weighted if there exists a set of weights \( w_i \) such that, for any number \( x \), it can be expressed as:

\[
x = \sum_{i=1}^{n} a_i w_i
\]

where \( a_i \) are a set of permissible digits. If the values of \( w_i \) are successive powers of the same number, the number system has a fixed base or a fixed radix, e.g. decimal system with base ten. Number systems in which the weights are not powers of the same number are called mixed-radix systems. Advantages of weighted number systems are the ease in performing magnitude comparison, sign detection and overflow detection.

A.2 Residue Number System

A.2.1 General Characteristics

The residue number system is an integer number system. Two important consequences from this property are: (1) quotients must generally be rounded to the closet integer and (2) in most cases the absolute value of the result will be larger than the input values used, hence rescaling is necessary. This implies division which is a slow process in this number system. In this system addition, subtraction and multiplication are inherently carry-free. This means that each digit of the result is a function of only one digit from each operand and independent of the others. Unlike division the above three operations can be performed very quickly, as in the case of multiplication the need for partial products is eliminated. Also division is not necessarily a closed operation in a residue system, that is certain stipulations are defined which allow for the inverse of a number to exist. The residue number system is not a weighted number system. Hence it does not have many of the advantageous properties listed for weighted number systems, such as magnitude comparison, sign detection and overflow detection.
A.2.2 Residue Representation

A residue number system can be completely represented by specifying its base. However unlike a fixed-radix number system, the base for residue numbers is not a single radix, but an N-tuple of integers $m_1, m_2, \ldots, m_n$ where each member is called a modulus. For any given base, the integer $x$ will have a residue representation that is also an N-tuple $\{r_1, r_2, \ldots, r_n\}$ where the $r_i$ are defined by a set of $N$ equations:

$$x = q_im_i + r_i \quad i = 1, 2, \ldots, N$$  \hspace{1cm} (A.2)

and $q_i$ is an integer chosen such that $0 \leq r_i < m_i$. $q_i$ can be thought of as the integer value of $\frac{x}{m_i}$ denoted as $\left[ \frac{x}{m_i} \right]$. The quantity $r_i$ is the least nonnegative integer remainder of the division of $x$ by $m_i$ designated as the residue of $x$ modulo $m_i$ or $|x|_{m_i}$. The integer $r_i$ is the $i$th residue digit of $x$. Note here that $x$ can have any sign, positive or negative, and $\left[ \frac{x}{m_i} \right]$ will have the same sign as $x$, but by definition $|x|_{m_i}$ must be nonnegative.

**Theorem A.1** Two integers $x$ and $x'$ have the same residue representation for moduli $m_1, m_2, \ldots, m_n$ if and only if $(x - x')$ is an integer multiple of the least common multiple of the moduli, denoted by $\overline{M}$. 


If we list the residue representation of a range of numbers (-4 to +32) for the set of three moduli (2, 3, and 5) two interesting features of the residue system are illustrated.

<table>
<thead>
<tr>
<th>Integers</th>
<th>Residue Digits Moduli</th>
<th>Integers</th>
<th>Residue Digits Moduli</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>-4</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>-3</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>-2</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>+2</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>+3</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>+4</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>+5</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>+6</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>+7</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>+8</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>+9</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>+10</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>+11</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>+12</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>+13</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>+14</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

(1) The residue representation is periodic. This means that if the representation is to be unambiguous in computation, there will be the restriction of using a single period, called the interval of definition.

(2) The residue representation does not appear to afford any easy means of performing magnitude comparison within any interval of definition, nor is the sign of a number easily apparent.
A.2.3 Representation of Negative Numbers

Similar to the binary system, the representation of negative number in residue arithmetic is somewhat arbitrary. One method is to represent the absolute magnitude of a number in residue code and use an external sign bit to represent the sign. Alternatively the sign of the number can be included within the residue code, similar to the complement representation in binary. It is common, for a dynamic range of \( M \), to consider residue numbers in the range of \([0, M/2 - 1]\) as positive and residue number in the range of \([M/2, M - 1]\) as negative.

Therefore if \( x \) is represented as \( \{r_1, r_2, \ldots, r_N\} \), \(-x\) is represented by

\[
\{(m_1 - r_1), (m_2 - r_2), \ldots, (m_N - r_N)\}
\]

A.2.4 Identities Involving Residue and Integer Values

A number of arithmetic relationships are presented here as a foundation for work discussed later in this chapter. Proof for the following identities can be found in [96].

**Identity 1**  Residues of multiples of \( m \)

\[
K \Theta_m m = 0 \quad \text{for } K \text{ an integer.}
\]  \hspace{1cm} (A.3)

The following observations can be made from the above identity:

1. \( |x|_1 = 0 \)
2. \[
\left\lfloor \frac{Kx}{Km} \right\rfloor = \left\lfloor \frac{x}{m} \right\rfloor \quad \text{for all integers } K
\]
3. \[
K \Theta_{Km} x = K|x|_m , \quad \text{for all integers } K
\]
4. \[
\left\lfloor \frac{a}{m} \right\rfloor = 0 \quad \text{if and only if } 0 \leq a < m
\]
5. \[
|a|_m = a \quad \text{if and only if } 0 \leq a < m
\]

**Identity 2**  Addition of Multiples of \( m \)

\[
x \Theta_m (\pm mK) = |x|_m
\]  \hspace{1cm} (A.4)
Again the following observations can be made from this identity:

\[
\left[ \frac{x \pm mK}{m} \right] = \left[ \frac{x}{m} \right] \pm K
\]

**Identity 3**  Additive Inverse modulo \( m \)

The following identities are derived from Eqn. (A.4) on page 130:

\[
| -x |_m = (m - 1) \Theta_m x = m \Theta_m (-x)
\]  \hspace{1cm} (A.5)

where \( |m - x|_m \) is called the additive inverse of \( x \) modulo \( m \). Every number has a unique additive inverse.

**Identity 4**  Addition and Subtraction modulo \( m \)

The following identities form the basis for addition and subtraction modulo \( m \):

\[
| x \pm y |_m = || x |_m \pm | y |_m |_m = | x |_m \pm | y |_m = x \Theta_m (\pm y)
\]  \hspace{1cm} (A.6)

where \( |x \pm y|_m \) is referred to as the sum or the difference of \( x \) and \( y \) modulo \( m \). This identity can be generalized to contain any number of terms:

\[
| \sum_{i=1}^{N} x_i |_m = \sum_{i=1}^{N} | x_i |_m
\]  \hspace{1cm} (A.7)

**Identity 5**  Multiplication Modulo \( m \)

As with addition and subtraction, the following identities form the basis for multiplication modulo \( m \):

\[
| xy |_m = | | x |_m y |_m | = | | x |_m y |_m = x \Theta_m y
\]  \hspace{1cm} (A.8)
A generalization of the above identity to include arbitrary number of terms is:

\[
\prod_{i=1}^{N} \alpha_{m} x_{i} = \prod_{i=1}^{N} \left| x_{i m} \right|
\]  

(A.9)

This identity holds for both negative and positive numbers. In the case of negative numbers, the additive inverse identity is used, which preserves the normal arithmetic laws.

**Identity 6**  
**Multiplicative Inverse**

Cancellation Law of Multiplication:

For \((K, m) = 1\) (greatest common divisor), if \(K \otimes_{m} a = K \otimes_{m} b\), then \(|a|_{m} = |b|_{m}\)

**Identity 7**  
**Existence of Multiplicative Inverse**

**Definition A.1**  
If \(0 \leq a < m\) and \(a \otimes_{m} b = 1\), \(a\) is called the multiplicative inverse of \(b\) mod \(m\), and is denoted by \(a = \left| b^{-1} \right|_{m}\).

**Theorem A.2**  
The quantity \(|b^{-1}|_{m}\) exists if and only if \((b, m) = 1\) and \(|b|_{m} \neq 0\).

In this case \(|b^{-1}|_{m}\) is unique.

**Theorem A.3**  
If the multiplicative inverse of \(b\) modulo \(m\), \(|b^{-1}|_{m}\), is \(|a|_{m}\), then

\(|a|^{-1}_{m} = |b|_{m}\)

**Theorem A.4**  
Fermat's Theorem

If \(p\) is a prime\(^1\), then

\(^1\) The Fermat-Euler Theorem is a generalization of this theorem and requires only that \(a\) and \(p\) be relatively prime.
\[ |a^p|_p = |a|_p \]  \hspace{1cm} (A.10)

Fermat's Theorem is important because it explicitly expresses the multiplicative inverse of $|a|_p$, recalling that $p$ is prime. The multiplicative inverse of $|a|_p$, for $|a|_p \neq 0$ is $|a^{p-2}|_p$ since $a^{p-2} \otimes_p a = 1$ by Fermat's Theorem. Hence an equation of the form $a \otimes_p x = |b|_p$ may be solved uniquely and the solution is:

\[ |x|_p = a^{-1} \otimes_p b = a^{p-2} \otimes_p b \]  \hspace{1cm} (A.11)

### A.3 Conversion Using the Chinese Remainder Theorem

The Chinese remainder theorem is a classical theorem from number theory which enables the conversion from the residue number system. Given the residue representation $\{r_1, r_2, ..., r_N\}$ of $x$, the Chinese remainder theorem makes it possible to determine $|x|_M$ provided the greatest common divisor of any pair of moduli is 1. Such moduli are called pairwise relatively prime.

**Theorem A.5**  \hspace{1cm} The Chinese Remainder Theorem

\[ |x|_M = \sum_{i=1}^{N} \otimes_{M_i} |r_i \otimes \hat{m}_i (\hat{m}_i)^{-1}| \hat{m}_i \]  \hspace{1cm} (A.12)

where $\hat{m}_i = \frac{M}{m_i}$ and $(m_j, m_k) = 1$ for $j \neq k$ and $M = \prod_{i=1}^{N} m_i$. From the Chinese Remainder Theorem $|x|_M$ is obtained, not $x$ itself. If $x$ lies in the range of 0 and $M-1$, then it can be written as:
\[ x = \sum_{i=1}^{N} \Theta_{\mathcal{G}} \hat{m}_{i} \left| r_{i} \otimes \hat{m}_{i} (\hat{m}_{i})^{-1} \right| \hat{m}_{i} \]  \hspace{1cm} (A.13)

since the modulo \( M \) operator on the left is not needed. Alternatively the Chinese Remainder Theorem can be written so that the sum appears without the modulo \( M \) operator. This can be done with the use of an auxiliary function \( A(x) \) shown below:

\[ x = \sum_{i=1}^{N} \hat{m}_{i} \left| r_{i} \otimes \hat{m}_{i} (\hat{m}_{i})^{-1} \right| \hat{m}_{i} - MA(x) \]  \hspace{1cm} (A.14)

\[ A(x) = \frac{1}{M} \left( \sum_{i=1}^{N} \hat{m}_{i} \left| r_{i} \otimes \hat{m}_{i} (\hat{m}_{i})^{-1} \right| \hat{m}_{i} - x \right) \]

where \( A(x) \) is a function of \( x \) defined for any integer \( x \). From Eqn. (A.14) it can be seen that \( A(x) \) is always an integer, and it can be shown that if \( 0 \leq x < M \) then:

\[ 0 \leq A(x) \leq \left( \frac{NM - \sum_{i=1}^{N} \hat{m}_{i}}{M} \right) \]

**A.3.1 Moduli with Common Factors**

If the moduli do not comply with the requirement that they be pairwise prime, then not every representation corresponds to a number. For example, for even moduli, even numbers have even residues and odd numbers have odd residues, therefore a residue representation containing both even and odd residues cannot exist if both moduli are even.
A.4 Simple Residue Arithmetic Operations

A.4.1 Addition and Subtraction

The addition and subtraction operation was presented earlier for one moduli. Based on that definition, the addition process for two numbers in residue representation can be derived.

Theorem A.6 Residue Addition Theorem

For a residue system consisting of moduli \( m_1, m_2, \ldots, m_N \) let \( x \) and \( y \) be represented in residue form. The residue representation of \( |x ± y|_M \) is:

\[
    x ± y \rightarrow (x \oplus m_1(\pm y), x \oplus m_2(\pm y), \ldots, x \oplus m_N(\pm y))
\]  

(A.15)

This theorem is a trivial result of the identity 4, but it bears fundamental importance in residue arithmetic. First, it shows that addition (or subtraction) has no intermodular carries (or borrows). In this respect residue superior to weighted number systems, since the absence of carries inherently results in higher speeds. Also, in weighted number systems in order to eliminate the carry propagation, extensive hardware is needed to implement carry look ahead logic. It can be argued that the hardware required in RNS for conversion replaces the additional hardware in a weighted number system. In residue this is accomplished without additional hardware. Second, the sum is obtained modulo \( M \), hence if the number exceeds \( M \), an ambiguity arises, since numbers of the form \(|a|_M \) and \(|a + kM|_M \) have the same residue representation, hence M must be chosen large enough to guarantee results within the dynamic range and to avoid overflow.

A.4.2 Multiplication in the Residue Representation

For the residue system consisting of the moduli \( m_1, m_2, \ldots, m_N \) let \( x \) and \( y \) be represented by residue digits. Then the residue representation of \(|x \otimes y|_M \) is
\[ |x \cdot y|_M \rightarrow (||x|_{m_1} \cdot |y|_{m_1})_m, ||x|_{m_2} \cdot |y|_{m_2})_m, \ldots, ||x|_{m_N} \cdot |y|_{m_N})_m \]  \hspace{1cm} (A.16)

Within the interval \([0, M-1]\), only one integer, namely \(|x \otimes y|_M\) has this residue representation. Multiplication like addition is also carry free and if \(x y\) exceeds \(M\) an ambiguity results from the periodic nature of the residue representation. An important aspect of multiplication is that it lends itself very well to table lookup. In conventional binary systems, table lookup for \(n\) bit wordlength would require \(2^{2n}\) entries. In residue however, for a comparable range, each modulus \(m_i\) requires \(m_i^2\) entries in the table and hence a total of \(\sum_{i=1}^{N} m_i^2\) entries are needed for all moduli.

### A.4.3 Definitions for Division in Residue Representation

Division in residue arithmetic can be classified into three categories:

4. **Division remainder zero**: Division where the dividend is known to be an integer multiple of the divisor and the divisor is known to be relatively prime.

5. **Scaling**: Division of an arbitrary dividend by any factor of \(M\) which is a product of \(m_i\)'s.

6. **General Division**: General division of an arbitrary integer by an arbitrary integer divisor.

Category two is analogous to power of two division in a binary system, where in the binary system this is accomplished by simply shifting the number. However in residue, it is not so simple, but nevertheless is much more faster than division by an arbitrary number. A more detailed description of category 3 can be found in [96]. Since scaling plays an important role in the structures developed later in this thesis, it will be described in further detail later in this chapter. Division Remainder Zero will be described here as it is closely related to other algorithms described later.
**Theorem A.7**  
Division Remainder Zero

\[
\left| \frac{b}{a|_{m_i}} \right| = a^{-1} \otimes_{m_m} b
\]  
(A.17)

for all \( m_i \), if and only if \( a \) divides \( b \) (without remainder) and \((a, m_i) = 1\)

### A.5 Conversions to the Residue Representation

#### A.5.1 Binary to Residue Conversion

Eqn. (A.2) defines the residue of a number modulo \( m_i \). In conventional computers, this calculation is performed by dividing \( x \) by \( m \) and determining the remainder. In a residue computer, which is capable of residue addition, multiplication, etc. a more efficient method can be used to determine the residue representation. A number is represented in the binary system as:

\[
x = 2^n b_n + \ldots + 2^2 b_2 + 2^1 b_1 + b_0
\]  
(A.18)

where \( b_i \) are the binary digits of the integer \( x \). Taking the modulo \( m_i \) yields:

\[
\left| x \right|_{m_i} = 2^n \otimes_{m_m} b_n \otimes_{m_m} \ldots \otimes_{m_m} 2^2 \otimes_{m_m} b_2 \otimes_{m_m} 2^1 \otimes_{m_m} b_1 \otimes_{m_m} b_0
\]  
(A.19)

If powers of 2 modulo \( m_i \) are directly available, \( \left| x \right|_{m_i} \) may be computed by merely adding (modulo \( m_i \)) those powers of 2 for which \( b_i = 1 \).

#### A.5.2 Mixed Radix Conversion

The Chinese Remainder Theorem is one method of converting residue numbers. The disadvantage of this method is the mod \( M \) operator, which would not make it feasible for residue machines which are designed to perform operation modulo \( m_i \). The mixed radix
conversion presented here, on the other hand, can be implemented in a residue machine, since it involves only \( \text{mod } m_i \) operations.

The mixed radix representation is of great importance in residue computation due to two reasons.

1) The mixed radix system is a weighted system and hence can be used in magnitude comparison.
2) Conversion from residue to certain mixed-radix systems is relatively fast in residue computers.

Before explaining the conversion procedure, it is necessary to explain the system itself.

**Mixed Radix System:**

A number \( x \) may be expressed in mixed radix form as:

\[
x = a_N \prod_{i=1}^{N-1} R_i + \ldots + a_3 R_2 R_1 + a_2 R_1 + a_1
\]  \hspace{1cm} (A.20)

where \( R_i \) are the radices and the \( a_i \) are the mixed radix digits and \( 0 \leq a_i < R_i \). For a given set of radices, the mixed radix representation of \( x \) is denoted \(^1\) by \( \langle a_N, a_{N-1}, \ldots, a_1 \rangle \) where the digits are in decreasing significance. It is obvious that a positive number in the interval \( \left[ 0, \prod_{i=1}^{N} R_i - 1 \right] \) may be represented uniquely in this manner. The multipliers of the mixed radix digits are the weights. For the special case of the decimal system the weights of the digits are consecutive powers of ten.

**Conversion to the Mixed-Radix System**

---

1. The procedure described in this section was first published by Garner [29]
If, for a set of moduli $m_1, m_2, \ldots, m_N$, a set of radices is chosen such that $m_i = R_i$, the mixed radix system and the residue system are said to be associated and the two systems have the same range of values, i.e. $\prod_{i=1}^{N} m_i$. If $m_i = R_i$, the mixed radix expression will be of the form:

$$x = a_N \prod_{i=1}^{N-1} m_i + \ldots + a_3 m_1 m_2 + a_2 m_1 + a_1$$  \hspace{1cm} (A.21)

where the $a_i$ are mixed radix coefficients. The coefficients are determined sequentially starting with $a_1$. Taking mod $m_1$ of Eqn. (A.21), will determine $a_1$, since all other terms except the last are multiples of $m_1$, therefore:

$$|x|_{m_1} = a_1$$  \hspace{1cm} (A.22)

Hence $a_1$ is simply the first residue digit. To obtain $a_2$, first the residue code of $x - a_1$ is formed. This quantity is divisible by $m_1$, and since $m_1$ is relatively prime to all other moduli, then the division by zero procedure on page 136 can be used to find the residue digits of order 2 to $N$ of $\frac{x-a_1}{m_1}$. From Eqn. (A.21), it can be deduced that $a_2 = \left| \frac{x-a_1}{m_1} \right|_{m_2}$.

In the same manner all the other mixed radix digits can be obtained. In general the mixed radix digits can be found for $i > l$ by:

$$a_i = \left| \frac{x}{m_1 m_2 \ldots m_{i-1}} \right|_{m_i}$$  \hspace{1cm} (A.23)
A.6 Extension of Base

Frequently it is necessary to find the residue representation of a number in one base based on its representation in another base. In most cases, the new base will be extension of the original base, with one or more extra moduli from the original base. The procedure, termed extension of base, is a mixed radix conversion with an additional final step. Consider a residue system consisting of moduli $m_1, m_2, \ldots, m_N$, and with the interval of definition $\left[0, \prod_{i=1}^{N} m_i - 1\right]$. If another modulus, $m_{N+1}$, is added to the base, then the interval of definition will become $\left[0, \prod_{i=1}^{N+1} m_i - 1\right]$ and the mixed radix expression will be of the form:

$$x = a_{N+1} \prod_{i=1}^{N} m_i + a_N \prod_{i=1}^{N-1} m_i + \ldots + a_3 m_1 m_2 + a_2 m_1 + a_1$$  \hspace{1cm} (A.24)

For any number in the original interval $a_{N+1}$ will be zero. and in performing the mixed radix conversion, this fact can be used to determine $|x|_{m_{N+1}}$.

A.7 Scaling

In conventional fixed-radix arithmetic, two commonly used operations are multiplication and division by a power of the base. This operation can be implemented easily in a digital computer by shifting the operand. Since shifting is fast, multiplication and division by a power of the radix offer obvious advantages over multiplying or dividing by an arbitrary number.

In terms of residue arithmetic, an analogy to fixed radix arithmetic would be division by a predetermined number that is a product of any of the moduli, which comprise the dynamic range $M$. Multiplication is not a consideration here since in residue, multiplication is a
simple operation, regardless of the multiplier. The division operation defined above is referred to as scaling.

A.7.1 Scaling Numbers

The method described here is generalized to include numbers of both positive and negative signs. In residue it is conventional to represent a negative number of magnitude $X$ as $M+X$. Division of any number can be represented as:

$$X = \left[\frac{X}{Y}\right]Y + |X|_Y$$  \hspace{1cm} (A.25)

where $X$ is the dividend and $Y$ is the divisor. The purpose of scaling is to find $\left[\frac{X}{Y}\right]$ for restricted values of $Y$. From Eqn. (A.25) $\left[\frac{X}{Y}\right]$ can be defined as:

$$\left[\frac{X}{Y}\right] = \frac{X - |X|_Y}{Y}$$  \hspace{1cm} (A.26)

Therefore the residue representation of $\left[\frac{X}{Y}\right]$ will be:

$$\left\{ \frac{|X - |X|_Y|}{Y}, \frac{|X - |X|_y|}{Y}, \ldots, \frac{|X - |X|_y|}{Y} \right\}_{m_1, m_2, \ldots, m_y}$$  \hspace{1cm} (A.27)

If $Y$ is a product of any of the moduli, then from Theorem A.7. for all $(m, Y) = 1$ one obtains:

$$\left[\frac{X - |X|_Y}{Y}\right]_{m_1} = \left[\frac{X}{Y}\right]_{m_1} = \left[\frac{X - |X|_Y}{Y}\right]_{m_1}$$  \hspace{1cm} (A.28)

1. The material in this section was first described in [107][95]
Eqn. (A.28) expresses all the residue digits of \( \frac{X}{Y} \) for which \((m_i, Y) = 1\). The rest of the digits can be obtained through base extension Section A.6

If it is known that the number is negative, then one can easily obtain \(X\) from \(M + X\). Scale by \(Y\) and represent the result as \(M + \left\lfloor \frac{X}{Y} \right\rfloor\). But if the sign of the number is not known, then if we scale the number as if it were positive, according to the above algorithm, the result would be \(\frac{M}{Y} + \left\lfloor \frac{X}{Y} \right\rfloor\). This can be avoided if we consider that division by \(Y\) maps all numbers in the interval \(\left[0, \frac{M}{2} - 1\right]\) into \(\left[0, \frac{M}{2Y} - 1\right]\) and all numbers in the range of \(\left[\frac{M}{2}, M\right]\) into \(\left[\frac{M}{2Y}, \frac{M}{Y}\right]\). Hence, it is possible to divide the number by \(Y\) first, then by noting the interval in which \(\left\lfloor \frac{X}{Y} \right\rfloor\) lies, determine the sign of \(X\). If \(X\) is negative add \(\left\lfloor \frac{-M}{Y} \right\rfloor\) to the result to obtain \(M + \left\lfloor \frac{X}{Y} \right\rfloor\). In this method of scaling, the result is rounded to the integer value closest but less than the actual answer. Methods that perform rounding to the closest integer are discussed in [96].

### A.8 Redundant Residue Number System (RRNS)

A redundant residue number system is defined as a residue system \(m_1, m_2, \ldots, m_N\) with \(r\) additional moduli. All \(r+N\) moduli must be relatively prime to ensure a unique number representation. The moduli \(m_1, m_2, \ldots, m_N\) are called the nonredundant moduli and the moduli \(m_{N+1}, m_{N+2}, \ldots, m_r\) the redundant moduli. A number in this system will be represented by \(N+r\) digits, \(N\) of which are nonredundant, and the rest redundant digits. The total range, the set of states represented by the RRNS, will be \([0, M_T - 1]\) where

\[
M_T = \prod_{i = 1}^{r} m_i.
\]

The interval \([0, M - 1]\) is termed the legitimate range where
\[ M = \prod_{i=1}^{N} m_i \] and the range \([M, M_T]\) termed the \textit{illegitimate} range. To make proper use of the redundancy, all operands and results must be restricted to the legitimate range. This constraint defines the dynamic range of the system to be \(\left[ -\frac{M-1}{2}, \frac{M-1}{2} \right] \) if \(M\) is odd and \(\left[ -\frac{M}{2}, \frac{M}{2} \right] \) if \(M\) is even. There exists a one-to-one correspondence between the integers in the dynamic range and the state of the legitimate range in nonredundant RNS. The mixed radix representations associated with the residue number states are used in both overflow detection and correction. Extensive literature on the subject of error and overflow detection can be found in [93].

\section{A.9 Complex Residue Number System}

So far the discussion on residue number systems has focused on definitions and properties defined over a finite ring or field, \(R(m)\) or \(F(p)\) if \(m=p\) a prime, in which the elements of this field/ring, \(M=\{0,...,m-1\}\) are real numbers. In this section, computations involving complex numbers in modular arithmetic will be introduced. A general description of complex residue number systems (CRNS) will be given, followed by special cases of the CRNS, which allow simplification in complex number computations [40].

Ordinary complex number systems are based on the fact that the equation \(x^2=-1\) has no solution in the set of real numbers. In order to permit solutions to this polynomial, the set of complex numbers is introduced, where \(j\), the imaginary unit, is equal to the square root of \(-1\). Analogous to this, in order to form a complex modular structure, it is necessary to first determine the solution of:

\[ x^2 = -1 \text{mod} m \quad (A.29) \]

If a solution to Eqn. (A.29) exists, then \(j \in R(m)\), and the equation is said to be solvable. In this case \(-1\) is a quadratic residue \textit{mod} \(m\). If it is otherwise, the equation is said to be
nonsolvable, and -1 is termed a quadratic nonresidue \( \text{mod } m \). The following theorem helps in determining whether the equation is solvable or not, for an arbitrary \( m \).

**Theorem A.8** The number -1 is a quadratic residue of all primes of the form \( p=4k+1 \) and a quadratic non residue of all primes of the form \( p=4k+3 \) [70].

If \( m \) is not a prime, then it is sufficient that -1 be quadratic residue of all the primes that divide \( m \) for there to be a solution, \( j \), of \( x^2 = -1(\text{mod} m) \). This is the first step in building a complex modular structure. The next step is to construct the complex extension field (for \( m \) prime) or complex extension ring (for \( m \) a non prime).

**Complex Extension Fields/Rings**

For the case \( m=p=4k+3 \). Eqn. (A.29) on page 143 has no solution in \( F(p) \). and \( j = \sqrt{-1} \notin F(p) \). A complex modular structure, with \( p^2 \) elements, isomorphic to the second degree Galois extension field \( F(p^2) \) can be formed by the ordered pairs \( (x_r, x_i) = x_r + jx_i \) with \( x_r, x_i \in F(p) \). The binary modular operations of addition and multiplication are defined as:

\[
(x_r, x_i) \oplus (y_r, y_i) = (u_r, u_i) \quad (A.30)
\]

\[
(x_r, x_i) \odot (y_r, y_i) = (z_r, z_i)
\]

where

\[
\begin{align*}
  u_r &= (x_r \oplus_p y_r) \\
  u_i &= (x_i \oplus_p y_i) \\
  z_r &= (x_r y_r \oplus_p (-x_i y_i)) \\
  z_i &= (x_i y_r \oplus_p (-x_r y_i))
\end{align*} \quad (A.31)
\]

It can be seen that complex modular operations emulate ordinary complex arithmetic, and similarly utilize four real multiplications and two real additions to perform complex multiplication.
If m is not a prime, then a complex modular ring \((R(m^2))\) can be formed by the ordered pairs defined above, following the same arithmetic rules. Work on this theme is presented in [7] and [37].

If \(m=p=4k+1\), then Eqn. (A.29) on page 143 is solvable. \(j \in R(m)\) and -1 is a quadratic residue \(mod\ m\). This case leads to the definition of a mapping from \(R(m^2)\) to a quadratic ring \(QR(m^2)\), which is isomorphic to \(R(m^2)\).

### A.10 Quadratic Like Residue Number System

The advantages which make the QRNS an attractive system for performing complex arithmetic, is that complexity of complex multiplication is reduced from four real multiplications to two real multiplications, and that real and imaginary data are mapped into two independent channels. The limitation of this system is the restriction placed on the moduli to be of the form \(4k+1\). Soderstrand [92] proposes a system that relaxes this restriction, at the cost of reduced resolution. The underlying concept here is to find a number in the RNS system that when squared yields a negative number. This residue number system is termed the Quadratic-Like Residue Number System (QLRNS), and retains the computational properties of QRNS. In QRNS \(j = \sqrt{-1}\), whereas in the QLRNS \(\sqrt{a} = j \sqrt{a}\), hence the resolution is reduced by the length of the \(j\) vector. This is an acceptable compromise: for example, in a 4-bit moduli set (16,15, 13, 11), the resolution is reduced from 15 bits to 12 bits in the imaginary term.

Complex number representation in the QLRNS proceeds in the following steps.

**Step 1.** Find integers \(m\) and \(n\) such that \(x + jy = m + nj \sqrt{a}\), where the approximation represents truncation or rounding to integer. Hence \(m = x\) and \(n = y \sqrt{a}\).
Step 2. \( j\sqrt{a} \) is now the real number \( j\sqrt{a} \) in QLRNS. Hence the complex number is represented by a pair of RNS numbers formed like complex conjugates:

\[
\begin{align*}
z &= m + n j\sqrt{a} \\
z^* &= m - n j\sqrt{a}
\end{align*}
\]

Step 3. A mapping \( f \) from the complex ring \((C(M))\) of complex RNS integers defined by \( m + nj\sqrt{a} \) to QLRNS can be defined by Eqn. (A.32). This mapping is invertible.\(^1\) The reverse mapping \( f^{-1} \) is defined by:

\[
\begin{align*}
m &= 2^{-1}(z + z^*) \\
n &= (2j\sqrt{a})^{-1}(z - z^*)
\end{align*}
\]

Observing the reverse mapping, the QLRNS can be further categorized into two subdivisions, one which the multiplicative inverses of \( 2 \) and \( 2j\sqrt{a} \) exist, and one that the inverses don't exist. For the latter case, the inverse mapping must be done using standard mixed radix conversion scaling techniques.

Complex arithmetic in QLRNS is defined by the following equations. Given two complex QLRNS number \((z_1, z_1^*)\) and \((z_2, z_2^*)\). addition, subtraction and multiplication are defined as:

\[
\begin{align*}
(z_1, z_1^*) \pm (z_2, z_2^*) &= (z_1 \pm z_2, z_1^* \pm z_2^*) \\
(z_1, z_1^*) \cdot (z_2, z_2^*) &= (z_1z_2, z_1^*z_2^*)
\end{align*}
\]

Eqn. (A.34) implies that complex multiplication in QLRNS can be performed with only two real multiplication.

---

1. The mapping from \( C(M) \) to QLRNS is an isomorphism. Proof of this can be found in [92]
As in the case of QRNS, this concept can be extended to both composite moduli and a system of quadratic rings using the direct sum mapping.

A.11 Modified Quadratic Residue Number System

In order to relax the restriction that exists for QRNS, namely that moduli be of the form of $4k+1$, a new number system is introduced that lifts this restriction at the cost of increasing the number of real multiplications involved in complex multiplication, from two to three. This method is termed the Modified Quadratic Residue Number System, or MQRNS for short [52][53][44][56]. This method, unlike the QLRNS method, does not result in a reduction in the dynamic range.

For moduli other than of the form $m=4k+1$, the monic equation $x^2 + 1 = 0$ is irreducible in $R(m)$. Therefore the QRNS method cannot be employed. In order to relax this restriction, the monic equation is generalized so that a solution other than $\sqrt{-1}$ exists over $R(m)$. An extension ring $MQR(m)$ is defined as:

$$MQR(m) = \{ \{ A^{(MQ)} \} : + . . \}$$  \hspace{1cm} (A.35)

The elements of this ring are defined by:

$$A^{(MQ)} = (A, A^*)$$  \hspace{1cm} (A.36)

where $A = |a + jb|_m$ and $A^* = |a - jb|_m$. $a, b \in R(m)$ and $A, A^* \in R(m)$ with $j$ as a solution to the monic quadratic $x^2 - n = 0$.

The binary operations modulo $m$ are calculated in the following manner:

Addition:
\[ A^{(MQ)} + B^{(MQ)} = (A + B, A^* + B^*) \] (A.37)

Multiplication:

\[ A^{(MQ)} \cdot B^{(MQ)} = \{(A \cdot B) - S, (A^* \cdot B^*) - S\} \] (A.38)

where \( S = \left( \frac{j^2}{m} \right) \oplus_m b \oplus_m d : h. d \) are the imaginary parts of the complex samples.

Since \( j^2 \) in \( m \neq -1 \), computation of the real and imaginary part of the product cannot be formed from the normal and conjugates terms, in the same manner as Eqn. (2.8) on page 23. Thus an alteration of the real component of the complex multiplication is required. In order to correct this value, \( S \) has to be calculated, which results in an extra multiplication over the QRNS method.

The real and imaginary parts of the product can be formed as:

\[
Y_{iR} = 2^{-1} \oplus_m (Q_i \oplus_m Q^*_i) \oplus_m (-S_i) \]

\[
Y_{iI} = 2^{-1} \oplus_m (Q_i \oplus_m Q^*_i) \] (A.39)

where \( Q_i = |A_i \cdot B_i| \) and \( Q^*_i = |A^*_i \cdot B^*_i| \). If \( S_i \) is subtracted directly from \( Q_i \) and \( Q^*_i \), then the real and imaginary parts of the complex product, can be computed from Eqn. (2.8) on page 23.

Once again using the isomorphism in Eqn. (A.40), it can be shown that computations over \( R(M) \) can be performed in \( L \) parallel rings.

\[
R(M) = MQR(m_1) \oplus MQR(m_2) \oplus \ldots \oplus MQR(m_L) \]

\[
M = \prod_{i=1}^{L} m_i \] (A.40)
Several works have been presented which demonstrate the use of MQRNS and QRNS in filter realizations [51][54][55].

A.12 Flexible Modulus RNS

The Flexible Modulus RNS (FMRNS) is an alternative to the QRNS method, which allows the use of any odd integer (>1) as modulus [114]. This flexibility relaxes the limitation on moduli, experienced with the QRNS, and allows implementation of greater dynamic ranges. The difference between the QRNS and FMRNS is that the polynomial \( g(X) \) used to generate the ideal in the quotient ring is not \( X^2 + 1 = 0 \) but, one that is merely "convenient". where \( X=j \). In order to effect the isomorphism between the direct product ring and the quotient ring, it is required that the difference of any two roots of \( g(X) \) be an invertible element of the ring \( R(m_k) \) for each \( k \). One such polynomial is

\[
g(X) = X(X^2 - 1)\]

This polynomial has three roots \( 0, +1, \) and \(-1\).

Defining \( FMR(m_k) = \{ S; \oplus, \otimes \}; \ S = \{ A^0, A^*, A^\dagger \} \) with \( A^0, A^*, A^\dagger \in R(m_k) \).

\( A^0 = a^\dagger, A^* = a^\dagger + a^\dagger, A^\dagger = a^\dagger - a^\dagger \). this is the evaluation of the polynomial \( a^\dagger + Xa^\dagger \) at the three roots. \( 0, +1, \) and \(-1\) of \( g(X) \). This evaluation map sets up an isomorphism between the quotient ring of polynomials modulo the ideal generated by \( g(X) \), and the cross-product of the ring \( R(m_k) \) with itself three times. Multiplication and additions are performed component-wise in these rings, just as in other cases.

The inverse mapping is comprised of the inverse of the polynomial map, followed by the Chinese Remainder Theorem, resulting in polynomials with coefficients in \( R(M) \). By replacing \( X \) with \( j \) the result in complex form is obtained.
Appendix B

Probability of Overflow
Error Calculation
Software

B.1 Modified Modulus C Program

/* C include files */

#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <time.h>

/* Macros for max., min. and ceil functions. */

#define max(a,b) (((a) > (b)) ? (a) : (b))
#define min(a,b) (((a) > (b)) ? (b) : (a))
#define ceil(a) ((int)(a+0.5))

/* File system variables: */

char sname[81]; /* Contains name of current file. */
char ename[81]; /* Contains name of current export file. */
char tname[81]; /* Contains name of current probability file. */
int saved; /* 1 if no changes since file last saved, 0 otherwise */
int named; /* 1 if model has received a filename, 0 otherwise. */

/* Distribution variables: */

int umaxval[4]; /* Index corresponds to stream. Contains max.
 value of uniform distribution for the stream. */
int uminval[4]; /* Index corresponds to stream. Contains min.
 value of uniform distribution for the stream. */
int ucheck[4]; /* Index corresponds to stream. 1 if unif distr.
 explicitly defined, 0 otherwise. */

int ncheck[4]; /* Index corresponds to stream. 1 if normal distr.
 explicitly defined, 0 otherwise. */
int absval[4]; /* Index corresponds to stream. Contains max absolu-
t value of approx. norm. distr. for the stream. */
int varval[4]; /* Index corresponds to stream. Contains variance of approx. norm. distr. for the stream. */

int dcheck[4]; /* Index corresponds to stream. 1 if distr. (unif. or normal) explicitly defined, 0 otherwise */
int startval[4]; /* Index corresponds to stream. Contains min. value of distr. for the stream. */
int endval[4]; /* Index corresponds to stream. Contains max. value of distr. for the stream. */
int maxbit[4]; /* Index corresponds to stream. Contains maximum power of two for which distr. has non-zero probability */
int dflag[4]; /* 0 if distribution for stream is uniform, 1 if it is normal. */

/* Variable choice variables: */

int v[4][16]; /* First index corresponds to stream, second to variable # (0=2, 1=4,...,7=256...) */
/* 0 if variable is not used, 1 if used */

int vcheck[4]; /* Index corresponds to stream. 1 if variables explicitly defined, 0 otherwise */
int vcheckbox;

/* Representation choice variables: */

int bit; /* Bit being processed in bit selection routine */

int rep[1200][16]; /* First index corresponds to representation #, second to variable #.
 Value is x if the representation employs the variable to the xth power, 0 otherwise.
 -1 for first variable denotes "extend prev. bit" */

char repstr[1200][51]; /* Index corresponds to rep. #. Contains symbolic rep. of bit. */

int bitrep[4][31][16]; /* First index corresponds to stream, second to bit, third to variable #.
 Contains bit representations. Values are as above. */

char bitrepstr[4][31][200]; /* First index corresp. to stream, second to bit. Contains symbolic rep. of bit. */

int bcheck[4]; /* Index corresponds to stream. 1 if bit rep. is explicitly defined, 0 otherwise. */

int brep[4][31]; /* First index corresponds to stream, second to bit. 
 Value is 1 if rep. defined for bit, 0 otherwise. */

/* Block length choice variables: */
int bl; /* Block length*/

/*Annealing variables.*/

int repfact,regmod, iters, iterstoend; /*Replication factor, required moduli, # of iterations,
 # of non-improving iterations to terminate*/

double decfactor, repwgt, aconf, atemp; /*Temperature decrement, weight of replication
factor in objective function.*/
desired confidence, initial temperature*/

int maxdegree[16]; /*Max. degree out output polynomials in each variable.*/

long totiters; /*Total # of iterations in annealing procedure.*/

/*Calculation variables: */

int mpow; /*Contains power of two represented by a term. */

int maxexp; /*Contains maximum degree of answer polynomials. Used to eliminate moduli too
small
to be used in inverse map. */

int ccheck; /* 0 if calculation not done. 1 otherwise */

int trnum; /* Counter used to cycle calcbutton to indicate calculation in progress. */

/*Information retrieval variables: */

FILE *f; /* Stream used to access the probability file. */

long indx[1024]; /* Array containing the offsets of the start of data for each term in
the probability file. */

int cj; /* # of the current term being processed (relative to prob. file order) */

int wj; /* # of the worst case term (relative to prob. file order) */

long curr; /* Contains offsets to data for current term in probability file. */

long wcurr; /* Same as above but for worst case bit*/

long mmod; /* Contains product of moduli to be used. */

double merr, perr; /* Contain error percentages. */

int labs; /*Contains least significant bit when finding max. error & */

int size; /*Contains size of largest moduli required, in bits */

int n; /*Contains number of moduli required */

int ckt; /*Contains type of moduli to use: 0=MRRNS, 1=QRNS, 2=Fermat. 3=Fermat &
Mersenne. */

/*Array of useable moduli. */

int mod[36][15]={( 7,5,3,-1,0,0,0,0,0,0,0,0,0,0,0,
15,13,11,7,-1,0,0,0,0,0,0,0,0,0,0,
3,29,27,25,23,19,17,13,11,7,-1,0,0,0,0,
63,61,59,55,53,47,43,41,37,31,29,23,19,17,13,
5,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
13,5,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,
29,25,17,13,-1,0,0,0,0,0,0,0,0,0,0,0,0,
61,53,41,37,29,25,17,13,5,-1,0,0,0,0,0,0,0,0,0,0
5. -1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
5. -1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
17.5. -1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
17.5. -1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
17.5. -1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
257.17.5. -1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
257.17.5. -1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
257.17.5. -1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
257.17.5. -1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
257.17.5. -1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
257.17.5. -1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
257.17.5. -1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
257.17.5. -1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
257.17.5. -1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
257.17.5. -1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
65537.257.17.5. -1.0.0.0.0.0.0.0.0.0.0.0.

7.5.3. -1.0.0.0.0.0.0.0.0.0.0.0.0.
7.5.3. -1.0.0.0.0.0.0.0.0.0.0.0.0.
31.17.7.5.3. -1.0.0.0.0.0.0.0.0.0.0.
31.17.7.5.3. -1.0.0.0.0.0.0.0.0.0.0.0.
127.31.17.7.5.3. -1.0.0.0.0.0.0.0.0.0.
257.127.31.17.7.5.3. -1.0.0.0.0.0.0.0.0.
257.127.31.17.7.5.3. -1.0.0.0.0.0.0.0.0.
257.127.31.17.7.5.3. -1.0.0.0.0.0.0.0.0.
257.127.31.17.7.5.3. -1.0.0.0.0.0.0.0.0.
257.127.31.17.7.5.3. -1.0.0.0.0.0.0.0.0.
257.127.31.17.7.5.3. -1.0.0.0.0.0.0.0.0.
8191.257.127.31.17.7.5.3. -1.0.0.0.0.0.0.
8191.257.127.31.17.7.5.3. -1.0.0.0.0.0.0.
8191.257.127.31.17.7.5.3. -1.0.0.0.0.0.0.
65537.8191.257.127.31.17.7.5.3. -1.0.0.0.0.0.

int cmod; /* Contains offset into mod array. Used to select MRNS, QRNS, Mersenne primes only. Fermat primes only. */

int go; /* Flag that is set to 1 once information has been retrieved. Enables PREV, NEXT, and WORST buttons */
char werrstr[20]; /* Contains error % of worst case term, in string form */
char wbstr[80]; /* Contains worst case term, in symbolic form */

/* General purpose and multi-use variables */

int a,b,c,d; /* Defines stream being processed. a = index of current stream */
/* b = index of other part of current input */
/* c = index of part of other input corresp. to part of current input */
/* d = index of part of other input not corresp. to part of current input */

int i,j; /* General purpose indexes */
char tempstr[200].tempstr2[200]; /* General purpose strings */
char *end; /* Dummy variable used in string conversions */

/* Initialization routine. Initializes all specification variables and flags to zero or null. Initializes filenames. */

int initialize()
time_t ticloc;
int i,j,k;

for (i=0;i<4;i++)
{
    for(k=0;k<16;k++)
    {
        v[i][k]=0;
    }
}

for (i=0;i<4;i++)
{
    vcheck[i]=0;
dcheck[i]=0;
uchck[i]=0;
ncheck[i]=0;
absval[i]=0;
varval[i]=0;
uminal[i]=0;
unaxval[i]=0;
maxbit[i]=0;
bccheck[i]=0;
}

saved=1;

for (i=0;i<4;i++)
    for (k=0;k<31;k++)
    {
        strcpy(bitrepstr[i][k],"\0");
        brep[i][k]=0;
        for (j=0;j<16;j++)
            bitrep[i][k][j]=0;
    }

for (i=0;i<4;i++)
{
    strcpy(bitrepstr[i][0],"2\0");
brep[i][0]=1;
    for (k=0;k<16;k++)
        bitrep[i][0][k]=0;
}

bl=0;
ccheck=0;
strcpy(sname,"noname.mod");
strcpy(tname,"temp.pgf");
mexp=0;
named=0;

atemp=100;
repfact=0;
reqmod=0;
iter=0;
decfactor=0.99;
repwgt=0.5;
aconf=0.99;
iterstoend=50;

srandom((int)time(&tloc));
return 0;

/* This procedure resets the bit representations of a stream and invalidates calculations. It is used if a change is made to distributions and/or variables after a representation has been defined (since representations depend on these specifications). */

int ResetBits(stream)
int stream;
{
int i,j;

bcheck[stream]=0;
for (i=1;i<31;i++)
{
    strcpy(bitrepstr[stream][i],"");
    brep[stream][i]=0;
    for (j=0;j<i6;j++)
        bitrep[stream][i][j]=0;
}
ccheck=0;
}

/* In order to work around a bug in the C PCW routine, this routine computes a to the bth power by repeated mult. */
int powr(a,b)
int a,b;
{
int temp,i;
temp=1;
for (i=1;i<=b;i++)
    temp=temp*a;
return temp;
}

/* This procedure returns the value of a group of bits of a value NUM, starting from the START th power of two, and ending at the END - 1 th power of two. */
int getbits(start,end,num)
int start,end,num;
{
int temp;
temp=(int)(abs(num)/powr(2,end));
temp=temp&(int)powr(2,start-end);
if (num<0)
    return temp-1;
else
    return temp;
}
/* Notify procedure for when the menu option to select variables for Stream A. Real Part is chosen. */

void var1_notify_proc()
{
    /* If changes would invalidate representations, confirm action. */
    if (bcheck[0]!=0)
    {
        ResetBits(0);
    }

    /* Set up indices to correct streams. This permits 1 procedure to handle all four streams. */
    a=0;
    b=1;
    c=2;
    d=3;

    /* Display the appropriate window, de-activate the variable menu button, and initialize title and choices in the newly displayed window. The ith choice is selected iff the ith bit in the checkbox value is 1. */

    vcheckbox=v[a][0]+2*v[a][1]+4*v[a][2]+8*v[a][3]+16*v[a][4]+
                32*v[a][5]+64*v[a][6]+128*v[a][7]+256*v[a][8]+512*v[a][9]+1024*v[a][10]+
                2048*v[a][11]+4096*v[a][12]+8192*v[a][13]+16384*v[a][14]+32768*v[a][15];
    printf("Variables: Stream A. Real Part: \
i/n", vcheckbox);

    return;
}

/* As above, but for Stream A. Imaginary Part. ????*/

void var2_notify_proc()
{
    if (bcheck[1]!=0)
    {
        ResetBits(1);
    }

    a=1;
    b=0;
    c=3;
    d=2;

    vcheckbox=v[a][0]+2*v[a][1]+4*v[a][2]+8*v[a][3]+16*v[a][4]+
                32*v[a][5]+64*v[a][6]+128*v[a][7]+256*v[a][8]+512*v[a][9]+1024*v[a][10]+
                2048*v[a][11]+4096*v[a][12]+8192*v[a][13]+16384*v[a][14]+32768*v[a][15];
    printf("Variables: Stream A. Imag. Part: \
i/n", vcheckbox);

    return;
}

/* As above, but for Stream B. Real Part */
void var1_notify_proc() {
    if (bcheck[2] != 0) {
        ResetBits[2];
    }
    a = 2;
    b = 3;
    c = 0;
    d = 1;
    vcheckbox = v[a][0] + 2*v[a][1] + 4*v[a][2] + 8*v[a][3] + 16*v[a][4] -
                32*v[a][5] + 64*v[a][6] + 128*v[a][7] + 256*v[a][8] + 512*v[a][9] + 1024*v[a][10] -
                2048*v[a][11] + 4096*v[a][12] + 8192*v[a][13] + 16384*v[a][14] - 32768*v[a][15];
    printf("Variables: Stream B, Real Part: %i n", vcheckbox);

    return;
}

/* As above, but for Stream B, Imaginary Part */

void var4_notify_proc() {
    if (bcheck[3] != 0) {
        ResetBits[3];
    }
    a = 3;
    b = 2;
    c = 1;
    d = 0;
    vcheckbox = v[a][0] + 2*v[a][1] + 4*v[a][2] + 8*v[a][3] + 16*v[a][4] -
                32*v[a][5] + 64*v[a][6] + 128*v[a][7] + 256*v[a][8] + 512*v[a][9] + 1024*v[a][10] -
                2048*v[a][11] + 4096*v[a][12] + 8192*v[a][13] + 16384*v[a][14] - 32768*v[a][15];
    printf("Variables: Stream B, Imag. Part: %i n", vcheckbox);

    return;
}

/* Notification procedure for when the menu option to define a representation for Stream A, Real Part */

void bit1_notify_proc() {
    /* If calculations have already occurred, redefining a representation will invalidate them, so confirm the choice. */

    /* Set up indices for the desired stream. */
    a = 0;
    b = 1;
    c = 2;
    d = 3;
/* Display the representation choice window, initialize the title of this window, and deactivate the Rep. button */
printf("Representation: Stream A, Real Part",NULL);

/* Initialize the selection procedure at bit 1, and print current selection. */
bit=1;
SetupBits(bit);
xv_set(bittext.PANEL_LABEL_STRING,"2^1",NULL);
xv_set(seltext.PANEL_LABEL_STRING,bitrepstr[a][1]);
return;
}

/* Same as above, but for Stream A, Imaginary Part. */

void bit2_notify_proc(menu, menu_item)
{
    Menu menu;
    Menu_item menu_item;
    {
        if (ccheck==1)
        {
            if (AskAboutCalc()==101)
                return;
            else
                ccheck=0;
        }
        a=1;
        b=0;
        c=3;
        d=2;
xv_set(bitbutton.PANEL_INACTIVE,TRUE,NULL);
xv_set(bitframe.XV_SHOW,TRUE,NULL);
xv_set(bitframe.FRAME_LABEL,"Representation: Stream A, Imag. Part",NULL);
        bit=1;
        SetupBits(bit);
        xv_set(bittext.PANEL_LABEL_STRING,"2^1",NULL);
        xv_set(seltext.PANEL_LABEL_STRING,bitrepstr[a][1]);
        return;
    }
}

/* Same as above, but for Stream B, Real Part */

void bit3_notify_proc(menu, menu_item)
{
    Menu menu;
    Menu_item menu_item;
    {
        if (ccheck==1)
        {
            if (AskAboutCalc()==101)
                return;
            else
                ccheck=0;
        }
        a=2;
        b=3;
        }
c=0;
d=1;
xy_set(bitbutton, PANEL_INACTIVE, TRUE, NULL);
xy_set(bitframe, XV_SHOW, TRUE, NULL);
xy_set(bitframe, FRAME_LABEL, "Representation: Stream B. Real Part", NULL);
bit=1;
SetupBits(bit);
xy_set(bittext, PANEL_LABEL_STRING, "2^1", NULL);
xy_set(seltext, PANEL_LABEL_STRING, bitrepstr[a][l]);
return;
}

/* Same as above, but for Stream B. Imaginary Part. */

void bit4_notify_proc(menu, menu_item)
{
    Menu menu;
    Menu_item menu_item;
    {
        if (ccheck==1)
            {
                if (AskAboutCalc()==101)
                    return;
                else
                    ccheck=0;
            }
    }
    a=3;
b=2;
c=1;
d=0;
xy_set(bitbutton, PANEL_INACTIVE, TRUE, NULL);
xy_set(bitframe, XSHOW, TRUE, NULL);
xy_set(bitframe, FRAME_LABEL, "Representation: Stream B. Imag. Part", NULL);
bit=1;
SetupBits(bit);
xy_set(bittext, PANEL_LABEL_STRING, "2^1", NULL);
xy_set(seltext, PANEL_LABEL_STRING, bitrepstr[a][l]);
return;
}

/* Notification procedure for when the menu choice to define a uniform distribution for stream A. Real Part is chosen. */

void unif1_notify_proc(menu, menu_item)
{
    Menu menu;
    Menu_item menu_item;
    {
        /* Confirm the choice if it will invalidate bit representations and/or calculations */
        if (bcheck[0]!=0)
            {
                if (AskAboutBits()==101)
                    return;
            }
    }
}
else
    ResetBits(0);
}

/* Set up indices for this stream. */
a=0;
b=1;
c=2;
d=3;

/* Initialize distribution file name, display uniform distribution definition
window, initialize this window's title and values, and deactivate the distribution button. */
xv_set(unifframe.XV_SHOW,TRUE,NULL);
xv_set(unifframe.FRAME_LABEL,"Uniform Distr:Stream A, Real Part",NULL);
xv_set(maxtext,PANEL_VALUE,umaxval[a],NULL);
xv_set(mintext,PANEL_VALUE,uminval[a],NULL);
xv_set(distribute,PANEL_INACTIVE,TRUE,NULL);
return;
}

/* Same as above but for Stream A, Imaginary Part */
void unif2_notify_proc(menu, menu_item):
    Menu menu;
    Menu_item menu_item;
    {
        if (bcheck[1]!=0)
            {
                if (AskAboutBits()==101)
                    return;
                else
                    ResetBits(1);
            }
        a=1;
b=0;
c=3;
d=2;
xv_set(unifframe.XV_SHOW,TRUE,NULL);
xv_set(unifframe.FRAME_LABEL,"Uniform Distr:Stream A, Imag. Part",NULL);
xv_set(maxtext,PANEL_VALUE,umaxval[a],NULL);
xv_set(mintext,PANEL_VALUE,uminval[a],NULL);
xv_set(distribute,PANEL_INACTIVE,TRUE,NULL);

        return;
    }

/* Same as above but for Stream B, Real Part */
void unif3_notify_proc(menu, menu_item)
    Menu menu;
    Menu_item menu_item;
    {
        if (bcheck[2]!=0)


160
```c
{
    if (AskAboutBits()==101)
        return;
    else
        ResetBits(2);
}

a=2;
b=3;
c=0;
d=1;
xv_set(unifframe.XV_SHOW,TRUE,NULL);
xv_set(unifframe.FRAME_LABEL,"Uniform Distr:Stream B, Real Part",NULL);
xv_set(imaxtext.PANEL_VALUE,umaxval[a],NULL);
xv_set(imintext.PANEL_VALUE,uminval[a],NULL);
xv_set(distrbutton.PANEL_INACTIVE,TRUE,NULL);
return;
}

/* Same as above, but for Stream B, Imaginary Part */

void unif4_notify_proc(menu, menu_item)
    Menu menu;
    Menu_item menu_item;
    {
        if (bcheck[3]!=0)
            {
                if (AskAboutBits()==101)
                    return;
                else
                    ResetBits(3);
            }
        a=3;
b=2;
c=1;
d=0;
xv_set(unifframe.XV_SHOW,TRUE,NULL);
xv_set(unifframe.FRAME_LABEL,"Uniform Distr:Stream B, Imag. Part",NULL);
xv_set(maxtext.PANEL_VALUE,umaxval[a],NULL);
xv_set(mintext.PANEL_VALUE,uminval[a],NULL);
xv_set(distrbutton.PANEL_INACTIVE,TRUE,NULL);
return;
}

/* Exactly as above, but for the normal distribution option, and for Stream A, Real Part. */

void norm1_notify_proc(menu, menu_item)
    Menu menu;
    Menu_item menu_item;
    {
        if (bcheck[0]!=0)
```
```c
{
    if (AskAboutBits() == 101)
        return;
    else
        ResetBits(0);
}

a=0;
b=1;
c=2;
d=3;
xv_set(normframe, XV_SHOW, TRUE, NULL);
xv_set(normframe, FRAME_LABEL, "Normal Distr: Stream A, Real Part", NULL);
xv_set(abtext, PANEL_VALUE, absval[a], NULL);
xv_set(vartext, PANEL_VALUE, varval[a], NULL);
xv_set(distributton, PANEL_INACTIVE, TRUE, NULL);
return;
}

/* Same as above but for Stream A, Imaginary Part */

void norm2_notify_proc(menu, menu_item)
{
    Menu menu;
    Menu_item menu_item;
    {
        if (bcheck[1] == 0)
            {
                if (AskAboutBits() == 101)
                    return;
                else
                    ResetBits(1);
            }
        a=1;
b=0;
c=3;
d=2;
xv_set(normframe, XV_SHOW, TRUE, NULL);
xv_set(normframe, FRAME_LABEL, "Normal Distr: Stream A, Imag. Part", NULL);
xv_set(abtext, PANEL_VALUE, absval[a], NULL);
xv_set(vartext, PANEL_VALUE, varval[a], NULL);
xv_set(distributton, PANEL_INACTIVE, TRUE, NULL);
    return;
}

/* Same as above, but for Stream B, Real Part */

void norm3_notify_proc(menu, menu_item)
{
    Menu menu;
    Menu_item menu_item;
    {
        if (bcheck[2] == 0)
            {
                if (AskAboutBits() == 101)
                    {
```
return;
else
   ResetBits(2);
}

a=2;
b=3;
c=0;
d=1;
xv_set(normframe,XV_SHOW,TRUE,NULL);
xv_set(normframe,FRAME_LABEL, "Normal Distr:Stream B, Real Part", NULL);
xv_set(abstext,PANEL_VALUE,absval[a],NULL);
xv_set(vartext,PANEL_VALUE, varval[a],NULL);
xv_set(distribution,PANEL_INACTIVE,TRUE,NULL);
return;
}

/* Same as above, but for Stream B, Imaginary Part. */

void norm4_notify_proc(menu, menu_item)
   Menu menu;
   Menu_item menu_item;
   {
     if (bcheck[3]!=0)
     {
       if (AskAboutBits()==101)
         return;
       else
         ResetBits(3);
     }
     a=3;
b=2;
c=1;
d=0;
xv_set(normframe,XV_SHOW,TRUE,NULL);
xv_set(normframe,FRAME_LABEL, "Normal Distr:Stream B, Imag. Part", NULL);
xv_set(abstext,PANEL_VALUE,absval[a],NULL);
xv_set(vartext,PANEL_VALUE, varval[a],NULL);
xv_set(distribution,PANEL_INACTIVE,TRUE,NULL);
return;
}

/* Notification procedure called when the information button is pressed. */

void infoproc (item,event)
   Panel_item item;
   Event event;
   {
     /* Open probability file */
f=fopen(tname,"rb");

     if (f==NULL)
       {
      notice_prompt(unifpanel,NULL,

/* Display information retrieval frame, initialize its values, and deactivate the information button. */
int go=0;
    xv_set(infoframe.XV_SHOW,TRUE,NULL);
    xv_set(infobutton.PANEL_INACTIVE,TRUE,NULL);
    xv_set(ichekbox.PANEL_VALUE,0,NULL);
    xv_set(lsbtext,PANEL_VALUE,1,NULL);
    xv_set(tchekbox,PANEL_VALUE,0,NULL);
    xv_set(ierrtext,PANEL_VALUE,",NULL);
    xv_set(ibittext,PANEL_VALUE,",NULL);
    xv_set(modtext,PANEL_VALUE,",NULL);
    xv_set(errtext,PANEL_VALUE,",NULL);
    xv_set(sizetext,PANEL_VALUE,",NULL);
    xv_set(numtext,PANEL_VALUE,",NULL);

    return ;
}

/* Notification procedure called when the block length menu option is chosen. */
void bienproc (item,event)
    Panel_item item;
    Event event;
{
    /* If changing the block length will invalidate calculations confirm choice. */
    if (ccheck==1)
    {
        if (AskAboutCalc()==101)
            return;
        else
            ccheck=0;
    }

    /* Display block length window, initialize values in the window, and deactivate the block length button. */
    xv_set(blenbutton,PANEL_INACTIVE,TRUE,NULL);
    xv_set(blenstart,PANEL_VALUE,bi,NULL);
    xv_set(blenframe.XV_SHOW,TRUE,NULL);
}

/* Notification procedure called when the OK button in the about box is pressed. */
void aboutokproc(menu, menu_item)
    Menuproc menu;
    Menu_item menu_item;
{
    /* Close About window and reactivate file button. */
```c
xv_set(aboutframe,XV_SHOW, FALSE, NULL);
xv_set(filebutton,PANEL_INACTIVE, FALSE, NULL);
return;
}

/* Notification procedure for when the Save-As menu option is chosen. */
void saveas_notify_proc(menu, menu_item)
    Menu menu;
    Menu_item menu_item;
{
    /* Display the Save window, initialize its values, and deactivate the file button. */
    xv_set(saveframe,XV_SHOW, TRUE, NULL);
    xv_set(savetext,PANEL_VALUE,sname,NULL);
    xv_set(filebutton,PANEL_INACTIVE, TRUE, NULL);
    return;
}

/* Notification procedure for when the Export button in the information window is pressed. */
void infoexportproc(menu, menu_item)
    Menu menu;
    Menu_item menu_item;
{
    /* Display the Export Window, initialize its values, and hide the information frame*/
    xv_set/exportframe,XV_SHOW, TRUE, NULL);
    xv_set/exporttext,PANEL_VALUE, "noname.txt", NULL);
    xv_set/exportframe,XV_SHOW, FALSE, NULL);
    return;
}

/* Notification procedure called when the Save menu option is chosen. */
void save_notify_proc(menu,menu_item)
    Menu menu;
    Menu_item menu_item;
{
    /* If the model is already saved, or has no name, nothing needs to be done.*/
    if (saved==1 || named==0)
        return;
    /*Save the model */
    save();
}
/* Notification procedure called when the Open menu option is chosen. */

void load_notify_proc(menu, menu_item)
{
    Menu menu;
    Menu_item menu_item;
    {
        /* If the current model has been changed since last being saved, confirm choice. */

        int answer;
        if (saved==0)
        {
            answer=notice_prompt(savepanel,NULL,
                    NOTICE_MESSAGE_STRINGS,
                    "Changes not saved. Continue?",NULL,
                    NOTICE_BUTTON,"YES",100,
                    NOTICE_BUTTON,"NO",101,
                    NULL);
            if (answer==101)
            {
                return;
            }
        }
    }

    /* Display the Open window, initialize its values, and deactivate the File button */

    xv_set(loadframe,XV_SHOW,TRUE,NULL);
    xv_set(loadtext,PANEL_VALUE,sname,NULL);
    xv_set(filebutton,PANEL_INACTIVE,TRUE,NULL);

    return;
}

/* Notification procedure called when the New menu option is chosen. */

void new_notify_proc(menu, menu_item)
{
    Menu menu;
    Menu_item menu_item;
    {
        /* If the current model has been changed since last being saved, confirm choice. */

        int answer;
        if (saved==0)
        {
            answer=notice_prompt(savepanel,NULL,
                    NOTICE_MESSAGE_STRINGS,
                    "Changes not saved. Continue?",NULL,
                    NOTICE_BUTTON,"YES",100,
                    NOTICE_BUTTON,"NO",101,
                    NULL);
            if (answer==101)
            {
                return;
            }
        }
    }
}
} /* Initialize all specifications. */
initialize();
activate();
return;
}

/* Notification procedure called when the OK button in the variable window is pressed. */

void varokproc (item, event)
Panel_item item;
Event_event event;
int i;

/* A change has occurred since the last save. */
saved=0;

/* Get the choices selected in the window */
for (i=0;i<16;i++)
{
    v[a][i]=getbits(i+1,i,INT)xv_get(vcheckbox,PANEL_VALUE));
}

/* Set flag indicating that variables have been defined for this stream. */
vcheck[a]=1;

/* If no variables have been defined for stream b, it will default to the values just defined for stream a. */
if (vcheck[b]==0)
    for(i=0;i<16;i++)
        v[b][i]=v[a][i];

/* If no vars have been defined for streams c or d, stream c will default to the values defined for stream a. */
if (vcheck[c]==0 && vcheck[d]==0)
    for(i=0;i<16;i++)
        v[c][i]=v[a][i];

/* If no vars have been defined for any of the other streams, stream d will default to the values defined for stream a. */
if (vcheck[d]==0 && vcheck[c]==0 && vcheck[b]==0)
    for(i=0;i<16;i++)
        v[d][i]=v[a][i];

/* Close the variable window and reactivate the variable button. */
xv_set(varframe,XV_SHOW,FALSE,NULL);
xv_set(varbutton,PANEL_INACTIVE,FALSE,NULL);

/* Activate the buttons based on the changes just made. */
activate();
}
/** Notification procedure called when the Cancel button in the variable window is pressed. */

void varcancelproc (item, event)
{
    Panel_item item;
    Event event;
    {
        /* Close the variable window and reactivate the variable button. */
        xv_set(varframe, XV_SHOW, FALSE, NULL);
        xv_set(varbutton, PANEL_INACTIVE, FALSE, NULL);
    }
}

/* Procedure that calculates the maximal number of bits required to represent the magnitude of START and END */

int maxb(start, end)
{
    int start, end;
    {
        int temp;
        temp=max(abs(start), abs(end));
        for (i=0; i<10; i++)
            if (powr(2,i)>temp)
                return i-1;
    }
}

/* Notification procedure called when the OK button in the uniform distribution window is pressed. */

void unifokproc(item, event)
{
    Panel_item item;
    Event event;
    {
        int i;
        /* A change has now occurred since the last save. */
        saved=0;

        /* Get the parameters of the distribution from the window. */
        uminval[a]=xv_get(mintext, PANEL_VALUE);
        umanxval[a]=xv_get(maxtext, PANEL_VALUE);

        /* Check for illegal values of these parameters. */
        if (uminval[0] > umanxval[0])
            {
                notice_prompt(unifpanel, NULL,
                NOTICE_MESSAGE_STRINGS,
}
“Max. value must be >= to min. value!”, NULL, NOTIFICATION_BUTTON,”OK”, 100, NULL);

    uminval[0]=0;
    return;
}

/* Set dflag to reflect normal distribution. */
dflag[a]=0;

/* Set startval, endval and maxbit based on the new information. */
startval[a]=uminval[a];
endval[a]=umaxval[a];
maxbit[a]=maxb(startval[a], endval[a]);
uchek[a]=1;
dcheck[a]=1;

/* Set defaults for other streams based on whether a uniform distribution has
been defined for these streams
and whether any distribution has been defined for these streams. (see varokproc
for details) */

if (uchek[b]==0)
{
    dflag[b]=dflag[a];
    uminval[b]=uminval[a];
    umaxval[b]=umaxval[a];
    if (dcheck[b]==0)
    {
        startval[b]=startval[a];
        endval[b]=endval[a];
        maxbit[b]=maxbit[a];
    }
}

if (uchek[c]==0 && ucheck[c]==0)
{
    dflag[c]=dflag[a];
    uminval[c]=uminval[a];
    umaxval[c]=umaxval[a];
    if (dcheck[c]==0 && dcheck[d]==0)
    {
        startval[c]=startval[a];
        endval[c]=endval[a];
        maxbit[c]=maxbit[a];
    }
}

if (uchek[d]==0 && ucheck[c]==0 && ucheck[b]==0)
{
    dflag[d]=dflag[a];
    uminval[d]=uminval[a];
    umaxval[d]=umaxval[a];
    if (dcheck[d]==0 && dcheck[c]==0 && dcheck[b]==0)
    {
        startval[3]=startval[a];
    }
}
endval[3]=endval[a];
maxbit[3]=maxbit[a];
}

/* Close uniform distribution window and reactivate distribution button. */
xv_set(uniframe,XV_SHOW,FALSE,NULL);
xv_set(distrbutton,PANEL_INACTIVE,FALSE,NULL);

/* Activate buttons based on changes just made. */
activate();

/* Notification procedure called when the Cancel button in the uniform distribution
window is pressed. */

void unifcancelproc (item,event)
    Panel_item item;
    Event event;
{
    /* Close uniform distribution window and reactivate distribution button. */
xv_set(uniframe,XV_SHOW,FALSE,NULL);
xv_set(distrbutton,PANEL_INACTIVE,FALSE,NULL);
}

/* This procedure uses 2-stage Simpson's rule to numerically integrate the normal
distribution from z=0.5 to z=0.5.
The mean of the distribution is 0, the variance is var. */

double norm(z,var)
    int z;
    int var;
{
    double b,a;/* Upper and lower limits of integration, resp. */
    int n; /* Number of panels. Note: n must be divisible by 4. */
    double i2n,i1n,istar;/* Integral values using n panels, n/2 panels, and using 2-
stage method. */
    int m,i,j,o,oo;
    double x,v,sum10,sum21,sum22,sum42,sum41;
    n=80;
a=z-0.5;
b=z+0.5;
sum41=0;
sum42=0;
sum21=0;
sum22=0;
sum10= 1/(2*sqrt(var*3.141592654)) *exp(-0.5*a*a/var) + 1/
(2*sqrt(var*3.141592654)) *exp(-0.5*b*b/var);
for (i=1;i<=n-1;i++)
{  
x=a+i/n;  
v=1/(2*sqrt(var*3.141592654)) *exp(-0.5*x*x/var);  
j=i/2;  
  
if (i%2==1) o=1;  
if (j%2==1) oo=1;  
  
if (o==1)      
    sum41=sum41+v;  
else  
    {  
      sum21=sum21-v;  
      if (oo==1)  
        sum42=sum42-v;  
      else  
        sum22=sum22-v;  
    }  
  
sum41=4*sum41;  
sum42=4*sum42;  
sum21=2*sum21;  
sum22=2*sum22;  
i2n=(sum10+sum41+sum21)/(3*n);  
iln=(sum10+sum42+sum22)/(1.5*n);  
  
istar=(16*i2n-iln)/15;  
  
return istar;  
}

/* Notification procedure called when the OK button in the normal distribution window is pressed. */
/* exactly the same logic as unifokproc except for the method used to calculation probabilities. */

void normokproc (item,event)  
  
  Panel_item item;  
  Event event;  
  {  
    int i,count;  
    double temp;  
    
    saved=0;  
    absval[a]=xv_get(absxtext,PANEL_VALUE);  
    varval[a]=xv_get(varxtext,PANEL_VALUE);  
    
    if (absval[a] <= 0)  
      {  
        notice_prompt(normpanel,NULL,  
          NOTICE_MESSAGE_STRINGS,"Invalid absolute value!",NULL,  
          NOTICE_BUTTON,"OK",100,  
          NULL);  
      }  

    saved=1;  
    
};
absval[a]=0;
return;
}

if (varval[a]<=0)
{
  notice_prompt(normpanel, NULL,
   NOTICE_MESSAGE_STRINGS, "Invalid variance!", NULL,
   NOTICE_BUTTON, "OK", 100,
   NULL);
  varval[a]=0;
  return;
}

startval[a]=-1*absval[a];
endval[a]=absval[a];
maxbit[a]=maxb(startval[a], endval[a]);

dflag[a]=1;
dcheck[a]=1;
nccheck[a]=1;

if (nccheck[b]==0)
{
  dflag[b]=dflag[a];
  absval[b]=absval[a];
  varval[b]=varval[a];
  if (dcheck[b]==0)
  {
    startval[b]=startval[a];
    endval[b]=endval[a];
    maxbit[b]=maxbit[a];
  }
}

if (nccheck[c]==0 && nccheck[d]==0)
{
  dflag[c]=dflag[a];
  absval[c]=absval[a];
  varval[c]=varval[a];
  if (dcheck[c]==0 && dcheck[d]==0)
  {
    startval[c]=startval[a];
    endval[c]=endval[a];
    maxbit[c]=maxbit[a];
  }
}

if (nccheck[d]==0 && nccheck[c]==0 && nccheck[b]==0)
{
  dflag[d]=dflag[a];
  absval[d]=absval[a];
  varval[d]=varval[a];
  if (dcheck[d]==0 && dcheck[c]==0 && dcheck[b]==0)
  {
    startval[d]=startval[a];
  }
}
endval[d]=endval[a];
maxbit[d]=maxbit[a];
}

xv_set(normframe,XV_SHOW,FALSE,NULL);
xv_set(distributton,PANEL_INACTIVE,FALSE,NULL);
activate();
}

/* Notification procedure called when the Cancel button in the normal distribution window is pressed.
   Same logic as unifcancel proc. */

void normcancelproc(item,event)
{
Panel_item item;
Event event;
{
xv_set(normframe,XV_SHOW,FALSE,NULL);
xv_set(distributton,PANEL_INACTIVE,FALSE,NULL);
}

/* Notification procedure called when the OK button in the block length window is pressed. */

void bieokproc(item,event)
{
Panel_item item;
Event event;
{
    /* A change has occurred since the last save. */
    saved=0;

    /* Get the new block length from the window. */
    bl=xv_get(blentext,PANEL_VALUE);

    /* Check for illegal values */
    if (bl<=0)
    {
        notice_prompt(blenpanel,NULL,
                NOTICE_MESSAGE_STRINGS, "Invalid Block Length!",NULL,
                NOTICE_BUTTON,"OK",100,
                NULL);
        return;
    }

    /* Close block length window and reactivate block length button */
    xv_set(blenframe,XV_SHOW,FALSE,NULL);
    xv_set(blenbutton,PANEL_INACTIVE,FALSE,NULL);

    /* Activate buttons based on changes just made. */
    activate();
}
}
/* Notification procedure called when the Cancel button in the block length window is pressed. */

void blencancelproc (item, event)
    Panel_item item;
    Event event;
{
    /* Close block length window and reactivate block length button */
    xv_set(blendframe, XV_SHOW, FALSE, NULL);
    xv_set(blendbutton, PANEL_INACTIVE, FALSE, NULL);
}

/* Notification procedure called when the OK button in the representation window is pressed. */

void bitokproc(item, event)
    Panel_item item;
    Event event;
{
    int i, j;

    "Check to see that ALL bits have received a representation."
    for (i=0; i<=maxbit[a]; i++)
        if (brep[a][i]!=1)
            {
                notice_prompt(unifpanel, NULL,
                NOTICE_MESSAGE_STRINGS,
                "Error! Representation not defined for all bits!".NULL,
                NOTICE_BUTTON,"OK",100,
                NULL);
                return ;
            }

    /* A change has occurred since the last save. */
    saved=0;

    /* Set flag indicating a representation has been defined for the stream. */
    bccheck[a]=1;

    /* If representations AND variables AND distributions have not been defined explicitly for the other streams,
    defaults occur as in normokproc, unifokproc, and varokpro. */

    if (bccheck[b]<=0 && bccheck[c]<=0 &&
        bccheck[d]<=0 && vcheck[b]==0 &&
        vcheck[c]==0 && vcheck[d]==0 &&
        dcheck[b]==0 && dcheck[c]==0 &&
        dcheck[d]==0)
    {
        maxbit[b]=maxbit[a];
        maxbit[c]=maxbit[a];
        maxbit[d]=maxbit[a];
        bccheck[b]=-1;/*Indicates a representation has been defined, but only by
    default. */
        bccheck[c]=-1;
bcheck[d]=-1;

for (i=0;i<=maxbit[a];i++)
{
    strcpy(bitrepstr[b][i], bitrepstr[a][i]);
    strcpy(bitrepstr[c][i], bitrepstr[a][i]);
    strcpy(bitrepstr[d][i], bitrepstr[a][i]);
    brep[b][i]=brep[a][i];
    brep[c][i]=brep[a][i];
    brep[d][i]=brep[a][i];
    for (j=0;j<16;j++)
    {
        bitrep[p][i][j]=bitrep[a][i][j];
        bitrep[c][i][j]=bitrep[a][i][j];
        bitrep[d][i][j]=bitrep[a][i][j];
    }
}

/* Close the representation window and reactivate the representation button. */
xv_set(bitframe,XV_SHOW,FALSE,NULL);
xv_set(bitbutton,PANEL_INACTIVE,FALSE,NULL);

/* Activate appropriate buttons based on changes. */
activate();
}

/* This procedure sets up a list of all the representations possible for the current bit,
in symbolic and flag array forms, and loads them into the representation list box. */
int SetupBits(bit)
int bit;
{
    int i1,i2,i3,i4,i5,i6,i7,i8,i9,i10,i11,i12,i13,i14,i15,i16,i,j,k,n;/ * Counters */

    int offset;/* Offset into list. */
    int count; /* # of different representations */

    int swap,aa,bb,temp[16];/* Used in bubble sort*/

    char *p;/* Pointer used in generating symbolic representations. */
    char tempstr[50];/* Used in generating symbolic (string) representations. */

    /* Current offset into the list of representations is 0. */
    offset=0;

    /* If the previous bit has received a representation specification, add the
    "Extend Prev. Bit"
    option to the list. */
    if (brep[a][bit-1]==1)
{ rep[0][0]=-1; /* -1 indicates bit extension */
  strcpy(restr[0], "Extend Prev. Bit");
  offset=1; /* Next item added to list will be offset by 1 to make room for
          the one just added. */
}
 /* 1 representation in list to date. */
count=offset;

/* Loop through all possible products of variables that could represent the current bit. */
for (i1=0; i1<min(bit, 18); i1++)
  for (i2=0; i2<min((bit-i1)/2, 18); i2++)
    for (i3=0; i3<min((bit-i1-i2)/3, 18); i3++)
      for (i4=0; i4<min((bit-i1-i2-i3)/4, 18); i4++)
        for (i5=0; i5<min((bit-i1-i2-i3-i4)/5, 18); i5++)
          for (i6=0; i6<min((bit-i1-i2-i3-i4-i5)/6, 18); i6++)
            for (i7=0; i7<min((bit-i1-i2-i3-i4-i5-i6)/7, 18); i7++)
              for (i8=0; i8<min((bit-i1-i2-i3-i4-i5-i6-i7)/8, 18); i8++)
                for (i9=0; i9<min((bit-i1-i2-i3-i4-i5-i6-i7-i8)/9, 18); i9++)
                  for (i10=0; i10<min((bit-i1-i2-i3-i4-i5-i6-i7-i8-i9)/10, 18); i10++)
                    for (i11=0; i11<min((bit-i1-i2-i3-i4-i5-i6-i7-i8-i9-i10)/11, 18); i11++)
                      for (i12=0; i12<min((bit-i1-i2-i3-i4-i5-i6-i7-i8-i9-i10-i11)/12, 18); i12++)
                        for (i13=0; i13<min((bit-i1-i2-i3-i4-i5-i6-i7-i8-i9-i10-i11-i12)/13, 18); i13++)
                          for (i14=0; i14<min((bit-i1-i2-i3-i4-i5-i6-i7-i8-i9-i10-i11-i12-i13)/14, 18); i14++)
                            for (i15=0; i15<min((bit-i1-i2-i3-i4-i5-i6-i7-i8-i9-i10-i11-i12-i13-i14)/15, 18); i15++)
                              for (i16=0; i16<min((bit-i1-i2-i3-i4-i5-i6-i7-i8-i9-i10-i11-i12-i13-i14-i15)/16, 18); i16++)
                                if (i1+i2+i3+i4+i5+i6+i7+i8+i9+i10>i11+i12+i13+i14+i15+i16) & &
                                  (i11=0 || v[a][0]=1) & & /* Check for a match and if all required
 variables are selected */
                                    { i2=0 || v[a][1]=1 } & & /* If so, copy the powers of the
                                    variables into */
                                    rep[count][0]=i1;
                                    /* the representation array */
                                    rep[count][1]=i2;
rep[count][2]=i3;
rep[count][3]=i4;
rep[count][4]=i5;
rep[count][5]=i6;
rep[count][6]=i7;
rep[count][7]=i8;
rep[count][8]=i9;
rep[count][9]=i10;
rep[count][10]=i11;
rep[count][11]=i12;
rep[count][12]=i13;
rep[count][13]=i14;
rep[count][14]=i15;
rep[count][15]=i16;
count=count+1;
}

/* Bubble sort the representations based on the max degree of the variables in
the terms. */
swap=1;
while (swap==1)
{
    swap=0;
    for (i=offset;i<count-1;i++)
    {
        aa=0;
        bb=0;
        for (j=0;j<16;j++) /* Find the max degree of variables in the terms. */
        {
            if (aa<rep[i][j])
                aa=rep[i][j];
            if (bb<rep[i+1][j])
                bb=rep[i+1][j];
        }
        if (aa>bb) /* If the terms are not in order, swap them. */
        {
            for (j=0;j<16;j++)
            {
                temp[j]=rep[i][j];
                rep[i][j]=rep[i+1][j];
                rep[i+1][j]=temp[j];
            }
            swap=1;
        }
    }
}

/* Loop through all representations, generating symbolic expressions for them. */
for(i=offset;i<count;i++)
{
    strcpy(repstr[i],""); /* Start with a null string. */
    if (rep[i][0]>0) /* If the variable 2 is used, */
    {
        /* i.e. non-zero power*/
to the string

    sprintf(tempstr,"%i",rep[i][0]); /* concatenate truncate 2^"power"
    strcat(repstr[i],"2""");
    strcat(repstr[i],tempstr);
    strcat(repstr[i],"*""");
}

if (rep[i][1]>0)
{
    sprintf(tempstr,"%i",rep[i][1]);
    strcat(repstr[i],"4""");
    strcat(repstr[i],tempstr);
    strcat(repstr[i],"*""");
}

if (rep[i][2]>0)
{
    sprintf(tempstr,"%i",rep[i][2]);
    strcat(repstr[i],"8""");
    strcat(repstr[i],tempstr);
    strcat(repstr[i],"*""");
}

if (rep[i][3]>0)
{
    sprintf(tempstr,"%i",rep[i][3]);
    strcat(repstr[i],"16""");
    strcat(repstr[i],tempstr);
    strcat(repstr[i],"*""");
}

if (rep[i][4]>0)
{
    sprintf(tempstr,"%i",rep[i][4]);
    strcat(repstr[i],"32""");
    strcat(repstr[i],tempstr);
    strcat(repstr[i],"*""");
}

if (rep[i][5]>0)
{
    sprintf(tempstr,"%i",rep[i][5]);
    strcat(repstr[i],"64""");
    strcat(repstr[i],tempstr);
    strcat(repstr[i],"*""");
}

if (rep[i][6]>0)
{
    sprintf(tempstr,"%i",rep[i][6]);
    strcat(repstr[i],"128""");
    strcat(repstr[i],tempstr);
    strcat(repstr[i],"*""");
}

if (rep[i][7]>0)
{
    sprintf(tempstr,"%i",rep[i][7]);
    strcat(repstr[i],"256""");
    strcat(repstr[i],tempstr);
    strcat(repstr[i],"*""");
}

if (rep[i][8]>0)
{
    sprintf(tempstr,"%i",rep[i][8]);
    strcat(repstr[i],"512""");
strcat(repstr[i], tempstr);
strcat(repstr[i], "***");
}

if (rep[i][9]>0)
{
    sprintf(tempstr, "%i", rep[i][9]);
    strcat(repstr[i], "1024***");
    strcat(repstr[i], tempstr);
    strcat(repstr[i], "***");
}

if (rep[i][10]>0)
{
    sprintf(tempstr, "%i", rep[i][10]);
    strcat(repstr[i], "2048***");
    strcat(repstr[i], tempstr);
    strcat(repstr[i], "***");
}

if (rep[i][11]>0)
{
    sprintf(tempstr, "%i", rep[i][11]);
    strcat(repstr[i], "4096***");
    strcat(repstr[i], tempstr);
    strcat(repstr[i], "***");
}

if (rep[i][12]>0)
{
    sprintf(tempstr, "%i", rep[i][12]);
    strcat(repstr[i], "8192***");
    strcat(repstr[i], tempstr);
    strcat(repstr[i], "***");
}

if (rep[i][13]>0)
{
    sprintf(tempstr, "%i", rep[i][13]);
    strcat(repstr[i], "16384***");
    strcat(repstr[i], tempstr);
    strcat(repstr[i], "***");
}

if (rep[i][14]>0)
{
    sprintf(tempstr, "%i", rep[i][14]);
    strcat(repstr[i], "32768***");
    strcat(repstr[i], tempstr);
    strcat(repstr[i], "***");
}

if (rep[i][15]>0)
{
    sprintf(tempstr, "%i", rep[i][15]);
    strcat(repstr[i], "65536***");
    strcat(repstr[i], tempstr);
    strcat(repstr[i], "***");
}

l=strlen(repstr[i]); /* Remove the trailing * from the string. */
p=repstr[i]+l-1;
*p=0;
}
n=(int)xv_get(bitlist,PANEL_LIST_NROWS); /* Find out how many items are already in the list box. */

for (i=0;i<n;i++)
    xv_set(bitlist,PANEL_LIST_DELETE,0,NULL); /* Delete them. */

for (i=count-1;i>=0;i--)
    xv_set(bitlist,PANEL_LIST_INSERT,0, /* Add the current representations. */
          PANEL_LIST_STRING,0,repstr[i],
          NULL);

    xv_set(bitlist,PANEL_LIST_SELECT,0,TRUE); /* Highlight the first representation. */

return 0;
}

/* Notification procedure that is called when the Previous button is pressed in the representation window. */

void bitprevproc(item, event)
    Panel_item item;
    Event event;
{
    /* If there is a previous bit, let that be the current bit, set up the representations, and display the current values for that bit. */
    if (bit>1)
        {
            bit=bit-1;
            SetupBits(bit);
            sprintf(tempstr,"%i",bit);
            strcpy(tempstr2,"2^^");
            strcat(tempstr2,tempstr);
            xv_set(bittext,PANEL_LABEL_STRING,tempstr2,NULL);
            xv_set(seltext,PANEL_LABEL_STRING,bitrepstr[a][bit],NULL);
        }
    return;
}

/* Notification procedure that is called when the Next button is pressed in the representation window. */

void bitnextproc(item, event)
    Panel_item item;
    Event event;
{
    /* If there is a next bit, let that be the current bit, set up the representations, and display the current values for that bit. */
    if (bit<maxbit[a])
        {
            bit=bit+1;
            SetupBits(bit);
            sprintf(tempstr,"%i",bit);
            strcpy(tempstr2,"2^^");
            strcat(tempstr2,tempstr);
            xv_set(bittext,PANEL_LABEL_STRING,tempstr2,NULL);
            xv_set(seltext,PANEL_LABEL_STRING,bitrepstr[a][bit],NULL);
        }
    return;
}
/*MARJ: Will need these functions*/

/* Procedure that takes two probability generating functions of random variables as input, and generates the probability generating function of the product of these random variables as output. */

int multrv(a, b, c, as, ae, bs, be, cs, ce)
/* a and b point to the start of the two input pgfs.
   'c points to the start of the output pgf.
   as,bs, and 'cs are the least exponents of the two input pgfs and the output pgf.
   ae,be, and 'ce are the greatest exponents of the two input pgfs and the output pgf. */

double 'a,'b,'c;
int as,ae,bs,be,'cs,'ce;
{

int i,j;
double 'temp;

/* Calculate greatest and least exponents of output pgf. */
'cs=min(min(as*bs,as*be), min(ae*bs, ae*be));
'ce=max(max(as*bs,as*be), max(ae*bs, ae*be));

/* Allocate memory for the output pgf. */
'c=(double *)malloc(("ce-'cs+1")*sizeof(double));

if ('c=NULL)
    {
        printf("Insufficient Memory!!");
        return 1;
    }

    temp='c; /* Simplify code by using a pointer, rather than a pointer to a pointer. */

/* Initialize output pgf to 0's (values are coeffs. of pgf) */
for (i=0;i<"cs-'cs;i++)
    temp[i]=0;

/* Calculate output pgf. */
for (i=as;i<ae;i++)
    for (j=bs;j<be;j++)
        temp[i*"cs]=temp[i*"cs]+a[i-as]*b[j-bs];

return 0;
}
/* Same as above, but output pgf is in this case the pgf of the sum of the two input random variables.*/

int multiply(a, b, c, as, ae, bs, be, cs, ce)
    double *a, *b, *c;
    int as, ae, bs, be, *cs, *ce;
{
    int i, j;
    double *temp;

    *cs = as + bs;
    *ce = ae + be;

    *c = (double *)malloc(("ce-"cs-1)*sizeof(double));
    if (*c == NULL)
        {
            printf("Insufficient Memory! ");
            return 1;
        }
    temp = *c;
    for (i = 0; i <= *ce - *cs; i++)
        temp[i] = 0;

    for (i = 0; i <= ae - as; i++)
        for (j = 0; j <= be - bs; j++)
            temp[i + j] = temp[i + j] + a[i] * b[j];

    return 0;
}

/* Calculates the pgf of -1 times a random variable.*/

int invert(b, a, bs, be, as, ae)
    /* a points to the input pgf, *b to the output pgf.
    as and *bs are the least exponents of the input and output pgfs
    ae and *be are the greatest exponents of the input and output pgfs*/
{
    double *a, *b;
    int as, ae, *bs, *be;
    
    double num;
    int i;
    double *temp;

    *bs = -1*ae;
*be=-1*as:

*b=(double *)malloc((*be-*bs+1)*sizeof(double));
temp=*b;

for (i=0;i<=(ae-as);i++)
{
    temp[i]=a[(ae-as)-i];
}

return 0;
}

" Same as above, plus a provision to ignore insignificant terms of the pgfs. ":

int multpolh(a, b, c, as, ae, bs, be, cs, ce, ao, bo, co)
/* Variables as before, but ao, bo, and *co are offsets into the pgfs. Values preceding these offsets are considered insignificant. */

double *a, *b, *c;
long as, ae, bs, be, *cs, *ce, ao, bo, *co;
{

int i, j;
double *temp;

"cs=as+bs+ao+bo;" Note change made due to offsets."
*ce=ae+be:

*c=(double *)malloc(*ce-*cs+1)*sizeof(double));

if (*c==NULL)
{
    printf("Insufficient Memory! ");
    return 1;
}

temp=*c;

for (i=0;i<=*ce-*cs;i++)
    temp[i]=0;

for (i=0;i<=ae-(as+ao);i++)
    for (j=0;j<=be-(bs+bo);j++)
        temp[i+j]=temp[i+j]+a[ao+i]*b[bo+j];

*co=0;

while (temp[*co]<0.0000000001)/" Set offset to ignore insignificant terms at the start of the output pgf. */
*co=co+1;

while(*temp[*ce-*cs]<0.0000000001) /* Simply reduce the value of the greatest
coeff. of the output pgf. to ignore*/
    *ce=ce-1;  /* Insignificant terms at its end. */
return 0;
}

/* Same as above, but optimized to calculate the sum of two indep. ident. distr. rand.
vars.
In this case, a points to the input pgf. */

int multsqr(a,c, as, ae,cs,ce,co,wo)
    double *a,**c;
    long as,ae,*cs,*ce,*wo,co;
{

    int i,j;
    double *temp;

    *cs=(as-co)^2;
    *ce=ae^2;

    *c=(double *)malloc(("ce-*cs+1)*sizeof(double));

    if (*"c==NULL)
        {
        printf("Insufficient Memory!");
            return 1;
        }

    temp=\"c;\n
    for (i=0;i<\"ce-*cs;i++)
        temp[i]=0;

    for (i=0;i<ae-(as+co);i++)
        {
            for (j=0;j<i;j++)
                temp[i+j]=temp[i+j]+2*a[co]*a[j+co];
            temp[i+i]=temp[i+i]+a[co]*a[co];
        }

    *wo=0;

    while (*temp[*wo]<0.0000000001)
        *
            while(*temp[*ce-*cs]<0.0000000001)
                *ce=ce-1;

184
return 0;
}

/* Same as multpolh but this procedure calculates the pgf of the DIFFERENCE of the two input rand. vars. */

int multpinv(a,b,c, as, ae, bs, be, cs, ce)
  double *a, *b, *c;
  long as, ae, bs, be, *cs, *ce;
{

  int i, j;
  double *temp;

  *cs=as-be;
  *ce=ae-bs;
  *c=(double *)malloc((*ce-*cs+1)*sizeof(double));

  if (*c==NULL)
    {
      printf("Insufficient Memory!\n");
      return 1;
    }

temp=*c;

  for (i=0; i<*ce-*cs; i++)
    temp[i]=0;

  for (i=as; i<=ae; i++)
    for (j=bs; j<=be; j++)
      temp[i-j-*cs]=temp[i-j-*cs]+a[i-as]*b[j-bs];

  return 0;
}

int redistr(a, pos, neg, as, ae)
  double **a, neg, pos;
  long *as, ae;
{

  int i, j;
  double *temp, *temp2;

ae = 1

temp = (double *) malloc((ae - as + 1) * sizeof(double));

if (temp == NULL)
{
    printf("Insufficient Memory!");
    return 1;
}

temp2 = "a;"

for (i = 0; i < ae - 1; i++)
    temp[i] = temp2[ae - i] * neg;

for (i = 0; i < ae; i++)
    temp[ae + i] = temp2[i] * pos;

free('a');

*a = temp;

return 0;
}

"MARJ: END Functions "

"MARJ: calculation procedure"/

" Main calculation procedure "/

int calculate()
{
    int t, cl, c2, swap; /* Used in bubble sort. */
    int s, i, j, k, l; /* General purpose counters. */

    int count[4]; /* Count of the number of terms in the input polynomials */

    FILE *hfo; /* Stream pointing to output probability file. */

    double *pgfl[4][31]; /* Array of pointers to the pgfs. of the coeffs. of the input polys.

        First index represents stream, second represents bit number*/

    int pgel[4][31], pgel[4][31]; /* Values of the least and greatest exponents in the

        pgfs.

        of the coeffs of the input polys. */

    double *pgf2[4][512]; /* Array of pointers to the pgfs of the coeffs of the pro-

        duct of the
input polys. First coeff as below, second coeff represents
represents bd.
2 represents cb. 3 represents ad */

int pgs2[4][512],pge2[4][512];/* Values of least and greatest exponents of the
pgfs of the product of input polys.*/

double *pgf3[2][1024];/* Array of pointers to the pgfs of the the coeffs of ac-
bd and ad+bc
First index indicates ac-bd or ad+bc (real or imag.). Second
indicates term # */

int pgs3[2][1024],pge3[2][1024];/* Values of the least and greatest exponents of
ac-bd and ad+bc*/

double *temp,*temp2;/* Temporary pointers.*/
double *pgf; /* Used to point to pgf of output coefficients. */
double *current; /* Points to pgf of an output coeff of dot product of block
length a power of two. */
double *working;/* Temporary pointer to a work pgf*/

long ps,pe,cs,ce,ws,we;/* Values of the least and greatest exponents in the
pgf, current, and working pgfs.*/
long offset,offsets,ceuoffsets;/* Values of the offsets for the pgf, current, and
working pgfs. */

int tempe,tempe2,temps,temps2;/* Temporary values to hold least and greatest
exponent values.*/
double num; /* Receives values from distribution files.*/
int bit[4][31]; /* First index corresponds to stream, second to bit group
#. Value is
least bit in the bit group. */

int offset,size,n,m;/*Temporary variables.*/

int found; /* Points to matches when comparing terms. */

int counter[4];/* Number of terms in product of input polys. Index as for pgf2*/
int counter2[2];/* Number of terms in ac-bd and ad+bc, respectively.*/

int pat[4][512][16];/* First index represents stream, second represents bit num-
ber. Holds term
representation of product of input polys in flag array
form.*/

int pat2[2][1024][2]; /* Holds term representation of ac-bd and ad+bc. First
index indicates real or imaginary
term, second indicates term number, third is # of term in
PAT*/

double neg[4],pos[4];/*Weight factors for positive and negative terms of pgfs*/
double prodpn[4];

/* For each stream...*/
for (s=0;s<4;s++)


/* Find the number of terms in the input polys and generate BIT, an array
 of the least bits in each bit group. */
    count[s]=0;
    for (i=0;i<=maxbit[s];i++)
    {
        if (bitrep[s][i][0] != -1)
        {
            bit[s][count[s]]=i;
            count[s]=count[s]+1;
        }
        bit[s][count[s]-maxbit[s]+1]; /* Used to stop loops later on. */
    }

/* Using the BIT array, determine the least and greatest coeff values of
 the terms of the input polys.
 and allocate memory for the pgfs of these coeffs. */
    for(i=0;i<count[s];i++)
    {
        size=powr(2.0,bit[s][i]+1)-bit[s][i]+1; /*MARJ*/
        pgfl[s][i]=(double *)malloc(size*sizeof(double));
        if (pgfl[s][i]==NULL)
        {
            printf("Insufficient Memory! \n");
            goto end;
        }

        /* Initialize the pgf to 0's */
        for (j=0;j<size;j++)
        {
            temp=pgfl[s][i];
            temp[j]=0;
        }

        /* Calculate least and greatest exponents of input pgfs. */
        pgel[0][i]=4;
        pgel[1][i]=4;
        pgel[2][i]=0;
        pgel[3][i]=0;
        pgs1[0][i]=-4; /*MARJ*/
        pgs1[1][i]=-4;
        pgs1[2][i]=0;
        pgs1[3][i]=0;
    }

    /* Read in the information from the distribution file. By isolating the
 bits of the value the probability read corresponds to, construct the pgfs of the input polys.
 */

    /*Uniform case.*/

    if (dflag[s]==0)
    {

for (j=0;j<count[s];j++)
{
    /*num=1.0/(pgel[s][j]-pgsl[s][j]+2);  MARJ */
    switch (s)
    {
        case 0:
            switch (j){
                case 0:
                {
                    temp=pgfl[0][0];
                    temp[0]=0.125*0.5;
                    temp[1]=0.125;
                    temp[2]=0.125;
                    temp[3]=0.125;
                    temp[4]=0.125;
                    temp[5]=0.125;
                    temp[6]=0.125;
                    temp[7]=0.125;
                    temp[8]=0.125*0.5;
                }
                break;
                case 1:
                {
                    temp=pgfl[0][1];
                    temp[0]=0.125*0.5;
                    temp[1]=0.125;
                    temp[2]=0.125;
                    temp[3]=0.125;
                    temp[4]=0.125;
                    temp[5]=0.125;
                    temp[6]=0.125;
                    temp[7]=0.125;
                    temp[8]=0.125*0.5;
                }
                break;
                case 2:
                {
                    temp=pgfl[0][2];
                    temp[0]=0.125*0.5;
                    temp[1]=0.125;
                    temp[2]=0.125;
                    temp[3]=0.125;
                    temp[4]=0.125;
                    temp[5]=0.125;
                    temp[6]=0.125;
                    temp[7]=0.125;
                    temp[8]=0.125*0.5;
                }
                break;
            }
            break;

        case 1:
        {
            temp=pgfl[1][j];
        }
    }
}
temp[0]=0;
}
break;

case 2:
{
    switch (j){
    case 0:
        {
            temp=pgf1[2][0];
            temp[0]=0.0371;
            temp[1]=0.0271;
            temp[2]=0.0708;
            temp[3]=0.0875;
            temp[4]=0.4809;
            temp[5]=0.0784;
            temp[6]=0.0775;
            temp[7]=0.0528;
            temp[8]=0.0379;
        }
        break;
    case 1:
        {
            temp=pgf1[2][1];
            temp[0]=0.3049;
            temp[1]=0.0270;
            temp[2]=0.0208;
            temp[3]=0.0239;
            temp[4]=0.5090;
            temp[5]=0.2252;
            temp[6]=0.0590;
            temp[7]=0.3164;
            temp[8]=0.0023;
        }
        break;
    case 2:
        {
            temp=pgf1[2][2];
            temp[0]=0.0009;
            temp[1]=0.0011;
            temp[2]=0.0012;
            temp[3]=0.0011;
            temp[4]=0.8810;
            temp[5]=0.0410;
            temp[6]=0.0076;
            temp[7]=0.0043;
            temp[8]=0.0018;
        }
        break;
    }
}
break;

case 3:
{
    temp=pgf1[3][j];
    temp[0]=0;
}
break;
/*for (i=0;i<=pqlt[s][0]-pgsl[s][0];i++)
   temp[i]=num;
 printf("j=%i   ,num=%f   , pgfl[s][j]=%f \\
\n",j,num,pgfl[s][j]);
} NARJ*/

"Set weight factors for positive or negative values."
/*
if (umaxval[s]<0)
{
   pos[s]=0;
   neg[s]=1;
}
else
   if (uminval[s]>0)
      {  
         pos[s]=1;
         neg[s]=0;
   }
else
      {  
         pos[s]=(double)umaxval[s]/(double)(umaxval[s]-uminval[s]);
         neg[s]=1-pos[s];
   }
 printf("pos[s]=%f   ,neg[s]=%f \\
\n",pos[s],neg[s]); NARJ"
}
else
   "Calculate the appropriate normal probability."
   {  
      for (j=0;j<count[s];j++)
         {
            for (i=0;i<=endval[s];i++)
               {  
                  num=norm(i,varval[s]);
                  temp=pgfl[s][j];
                  offset=getbits(bit[s][j-1],bit[s][j],i);
                  temp[offeet-pgsl[s][j]]=
                  temp[offeet-pgsl[s][j]]+num;
               }
            pos[s]=0.5;
            neg[s]=0.5;
         }
   }
   /* In case errors occurred in the generation of the distribution files, make sure the sum of the coeffs of pgfs is 1 by scaling them.*/
   /*for (j=0;j<count[s];j++)
   {  
      num=0;
temp=pgfl[s][j];
for (i=pgsl[s][j];i<=pgel[s][j];i++)
    num+=temp[i-pgsl[s][j]];
for (i=pgsl[s][j];i<=pgel[s][j];i++)
    temp[i-pgsl[s][j]]=temp[i-pgsl[s][j]]/num;
printf("\%d,\%d,\%d,pgfl[\%d]\n","i","count[\%d","pgfl[\%d]"

); // MARJ */

/* Calculate the pg's of the coeff's of the terms of ac. */
counter[0]=0;
for (i=0;i<count[0];i++) /* loop through all terms of a */
    for (j=0;j<count[2];j++) /* loop through all terms of c */
    {
        found=-1; /* Search to see if a term with the form of the product of the */
        for (k=0;k<count[0];k++) /* two current terms already exists. */
            if (pat[0][k][0]
                && pat[0][k][1]
                && pat[0][k][2]
                && pat[0][k][3]
                && pat[0][k][4]
                && pat[0][k][5]
                && pat[0][k][6]
                && pat[0][k][7]
                && pat[0][k][8]
                && pat[0][k][9]
                && pat[0][k][10]
                && pat[0][k][11]
                && pat[0][k][12] ==bitrep[0][bit[0][i][0]-
                       bitrep[2][bit[2][j][0]-
                       bitrep[0][bit[0][i][1]-
                       bitrep[2][bit[2][j][1]-
                       bitrep[0][bit[0][i][2]-
                       bitrep[2][bit[2][j][2]-
                       bitrep[0][bit[0][i][3]-
                       bitrep[2][bit[2][j][3]-
                       bitrep[0][bit[0][i][4]-
                       bitrep[2][bit[2][j][4]-
                       bitrep[0][bit[0][i][5]-
                       bitrep[2][bit[2][j][5]-
                       bitrep[0][bit[0][i][6]-
                       bitrep[2][bit[2][j][6]-
                       bitrep[0][bit[0][i][7]-
                       bitrep[2][bit[2][j][7]-
                       bitrep[0][bit[0][i][8]-
                       bitrep[2][bit[2][j][8]-
                       bitrep[0][bit[0][i][9]-
                       bitrep[2][bit[2][j][9]-
                       bitrep[0][bit[0][i][10]-
                       bitrep[2][bit[2][j][10]-
                       bitrep[0][bit[0][i][11]-
                       bitrep[2][bit[2][j][11]-
                       bitrep[0][bit[0][i][12]+" bitrep[2][bit[2][j][12]+" bitrep[0][bit[0][i][13]+" bitrep[2][bit[2][j][13]+" bitrep[0][bit[0][i][14]+" bitrep[2][bit[2][j][14]*/

192

found=k;

printf("i=%i ,j=%i ,k=%i ,pat[0][k][0]=%i
 .pat[0][k][1]=%i ,pat[0][k][2]=%i ,pat[0][k][3]=%i ,pat[0][k][4]=%i
 .pat[0][k][5]=%i ,pat[0][k][6]=%i ,pat[0][k][7]=%i ,pat[0][k][8]=%i
 .pat[0][k][9]=%i ,pat[0][k][10]=%i ,pat[0][k][11]=%i ,pat[0][k][12]=%i
 .pat[0][k][13]=%i ,pat[0][k][14]=%i ,pat[0][k][15]=%i \n", i, j, k,
pat[0][k][0], pat[0][k][1], pat[0][k][2], pat[0][k][3], pat[0][k][4], pat[0][k][5], pat[0][k][6], pat[0][k][7],
pat[0][k][8], pat[0][k][9], pat[0][k][10], pat[0][k][11], pat[0][k][12], pat[0][k][13], pat[0][k][14], pat[0][k][15]);

if (found==1) /* If not, calculate the pgf of coeff of the */
{
  /* product of the two terms */
  if (mult((pgfl[0][i], pgfl[2][j]),
           &pgf2[0][counter[0]],
           pgs[0][i], pgs[0][j],
           pgsl[2][j], pgsl[0][i],
           &pgs2[0][counter[0]],
           &pgs2[0][counter[0]])) == 1)
    goto end; /* (goto terminates calculations in case of insufficient memory) */

  for (l=0; l<i6; l++) /* Insert new product term in PAT array. */
  {
    pat[0][counter[0]][0] = bitrep[0][bit[0][i]][1] =
    bitrep[2][bit[2][j]][1];
    counter[0] = counter[0] + l;

    /*print("i=%i ,j=%i ,k=%i ,l=%i
 .pat[0][counter[0]][0]=%i ,pat[0][k][1]=%i ,pat[0][k][2]=%i
 .pat[0][k][3]=%i ,pat[0][k][4]=%i ,pat[0][k][5]=%i ,pat[0][k][6]=%i
 .pat[0][k][7]=%i ,pat[0][k][8]=%i ,pat[0][k][9]=%i ,pat[0][k][10]=%i
 .pat[0][k][11]=%i ,pat[0][k][12]=%i ,pat[0][k][13]=%i ,pat[0][k][14]=%i
 .pat[0][k][15]=%i \n", i, j, k, l,
    pat[0][counter[0]][0], pat[0][counter[0]][1], pat[0][counter[0]][2], pat[0][counter[0]][3], pat[0][counter[0]][4], pat[0][counter[0]][5], pat[0][counter[0]][6], pat[0][counter[0]][7],
    pat[0][counter[0]][8], pat[0][counter[0]][9], pat[0][counter[0]][10], pat[0][counter[0]][11], pat[0][counter[0]][12], pat[0][counter[0]][13], pat[0][counter[0]][14], pat[0][counter[0]][15]) ; */
  }
}

else
{
if (multiply(pgf1[0][i].pgf2[0][j].pgf1[2][j].pgf2[0][j].pgf1[2][j]) /* If a term of the same
form as the current product term*/
    temp, /* already exists, calculate
the pgf of the coeff of the*/
    pgs1[0][i].pgf1[0][i]./* product term, then multi-
ply this pgf with that of the*/
    pgs1[2][j].pgf1[2][j]./* pre-existing term.*/
    &temp,
    &tempe)==l) goto end;

if (multiply(temp.pgf2[0][found].
    &temp2,
    tems.tempe,
    pgs2[0][found].
    pgs2[0][found],
    &temp2,
    &tempe2)==l) goto end;

pgs2[0][found]=temps2; /* Set least and greatest exponents
of the new pgf of the*/
unneeded pgfs; /*

free(pgf2[0][found]);
pgf2[0][found]=temp2;
free(temp);
}

"Same as above, but for bd"/

counter[1]=0;
for (i=0;i<counter[1];i++)
    for (j=0;j<counter[3];j++)
    {
        found=1;
        for (k=0;k<counter[1];k++)
            if (pat[1][k][0]
                ==bitrep[1][bit[1][i][0]+
                bitrep[3][bit[3][j]][0]
                && pat[1][k][1]
                ==bitrep[1][bit[1][i][1]+
                bitrep[3][bit[3][j]][1]
                && pat[1][k][2]
                ==bitrep[1][bit[1][i][2]+
                bitrep[3][bit[3][j]][2]
                && pat[1][k][3]
                ==bitrep[1][bit[1][i][3]+
                bitrep[3][bit[3][j]][3]
                && pat[1][k][4]
                ==bitrep[1][bit[1][i][4]+
                bitrep[3][bit[3][j]][4]
                && pat[1][k][5]
                ==bitrep[1][bit[1][i][5]+
                bitrep[3][bit[3][j]][5]
& & pat[1][k][6] 
  ==bitrep[1][bit[1][i]][6] + 
    bitrep[3][bit[3][j]][6] 
& & pat[1][k][7] 
  ==bitrep[1][bit[1][i]][7] + 
    bitrep[3][bit[3][j]][7] 
& & pat[1][k][8] 
  ==bitrep[1][bit[1][i]][8] + 
    bitrep[3][bit[3][j]][8] 
& & pat[1][k][9] 
  ==bitrep[1][bit[1][i]][9] + 
    bitrep[3][bit[3][j]][9] 
& & pat[1][k][10] 
  ==bitrep[1][bit[1][i]][10] + 
    bitrep[3][bit[3][j]][10] 
& & pat[1][k][11] 
  ==bitrep[1][bit[1][i]][11] + 
    bitrep[3][bit[3][j]][11] 
& & pat[1][k][12] 
  ==bitrep[1][bit[1][i]][12] + 
    bitrep[3][bit[3][j]][12] 
& & pat[1][k][13] 
  ==bitrep[1][bit[1][i]][13] + 
    bitrep[3][bit[3][j]][13] 
& & pat[1][k][14] 
  ==bitrep[1][bit[1][i]][14] + 
    bitrep[3][bit[3][j]][14] 
& & pat[1][k][15] 
  ==bitrep[1][bit[1][i]][15] + 
    bitrep[3][bit[3][j]][15] 
found=k;

if (found==1)
{
  if (multrv(pgfl1[i], pgfl3[j],
    spgf2[1][counter1],
    pg1[i], pgel1[i][i],
    pg3[j], pgel3[j][j],
    &pgs2[1][counter1],
    &pgel2[1][counter1])==1) goto end;

  for (l=0; l<16; l++)
    pat[1][counter1][l]=
    bitrep[1][bit[1][i]][l] +
    bitrep[3][bit[3][j]][l];
  counter1=counter1+1;
}
else
{
  if (multrv(pgfl1[i], pgfl3[j],
    &temp,
    pg1[i], pgel1[i][i],
    pg3[j], pgel3[j][j],
    &temps,
    &tempe)==1) goto end;

  if (multpoly(temp, pgf2[1][found],
    &temp2,
    temps, tempe,
pgs2[1][found],
pge2[1][found],
&tempe2,
&tempe2==1) goto end;

pgs2[1][found]=temps2;
pge2[1][found]=tempe2;
free(pgf2[1][found]);
pgf2[1][found]=temp2;
free(temp);

/

Same as above but for bc */

counter[2]=0;
for (i=0;i<count[1];i++)
  for (j=0;j<count[2];j++)
    {
      found=1;
      for (k=0;k<counter[2];k++)
        if (pat[2][k][0]
          ==bitrep[1][bit[1][i]][0]+bitrep[2][bit[2][j]][0])
          & pat[2][k][1]
          ==bitrep[1][bit[1][i]][1]+bitrep[2][bit[2][j]][1]
          & pat[2][k][2]
          ==bitrep[1][bit[1][i]][2]+bitrep[2][bit[2][j]][2]
          & pat[2][k][3]
          ==bitrep[1][bit[1][i]][3]+bitrep[2][bit[2][j]][3]
          & pat[2][k][4]
          & pat[2][k][5]
          ==bitrep[1][bit[1][i]][5]+bitrep[2][bit[2][j]][5]
          & pat[2][k][6]
          & pat[2][k][7]
          & pat[2][k][8]
          ==bitrep[1][bit[1][i]][8]+bitrep[2][bit[2][j]][8]
          & pat[2][k][9]
          ==bitrep[1][bit[1][i]][9]+bitrep[2][bit[2][j]][9]
          & pat[2][k][10]
          ==bitrep[1][bit[1][i]][10]+bitrep[2][bit[2][j]][10]
          & pat[2][k][11]

found=k;
if (found!=-1)
{
  if (multrv(pgf1[1][i], pgf1[2][j], ipgf2[2][counter[2]], 
        pgs1[1][i], pgs1[1][i], 
        pgs1[2][j], pgs1[2][j], 
        ipgs2[2][counter[2]], 
        ipgs2[2][counter[2]])==1)
    goto end;

  for (l=0;l<16;l++)
  {
    pat[2][counter[2]][1]=
      bitrep[1][bit[1][i]][1]+ bitrep[2][bit[2][j]][1];
  }
}
else
{
  if (multrv(pgf1[1][i], pgf1[2][j], 
            temp, 
            pgs1[1][i], pgs1[1][i], 
            pgs1[2][j], pgs1[2][j], 
            temps, 
            tempe)==1) goto end;

  if (multpoly(temp, pgf2[2][found], 
                temp2, 
                temps, tempe, 
                pgs2[2][found], 
                pge2[2][found], 
                temps2, 
                tempe2)==1) goto end;

  pgs2[2][found]=temps2;
  pge2[2][found]=tempe2;
  free(pgf2[2][found]);
  pgf2[2][found]=temp2;
  free(temp);
}
}
/* Same as above but for ad*/

counter[3]=0;
for (i=0;i<count[0];i++)
    for (j=0;j<count[3];j++)
    {
        found=1;
        for (k=0;k<counter[3];k++)
            if (pat[3][k][0]
                ==bitrep[0][bit[0][i]][0]
                & pat[3][k][1]
                ==bitrep[0][bit[0][i]][1]
                & pat[3][k][2]
                ==bitrep[0][bit[0][i]][2]
                & pat[3][k][3]
                ==bitrep[0][bit[0][i]][3]
                & pat[3][k][4]
                ==bitrep[0][bit[0][i]][4]
                & pat[3][k][5]
                ==bitrep[0][bit[0][i]][5]
                & pat[3][k][6]
                ==bitrep[0][bit[0][i]][6]
                & pat[3][k][7]
                ==bitrep[0][bit[0][i]][7]
                & pat[3][k][8]
                ==bitrep[0][bit[0][i]][8]
                & pat[3][k][9]
                ==bitrep[0][bit[0][i]][9]
                & pat[3][k][10]
                ==bitrep[0][bit[0][i]][10]
                & pat[3][k][11]
                ==bitrep[0][bit[0][i]][11]
                & pat[3][k][12]
                ==bitrep[0][bit[0][i]][12]
                & pat[3][k][13]
                ==bitrep[0][bit[0][i]][13]
                & pat[3][k][14]
                ==bitrep[0][bit[0][i]][14]
                & pat[3][k][15]
                ==bitrep[0][bit[0][i]][15]
            )
        found=k;
if (found==-1)
{
    if (mulcrv(pgf1[0][i],pgf1[3][j],
            ppgf2[3][counter[3]],
            pgs1[0][i],pgs1[0][i],
            pgs1[3][j],pgs1[3][j],
            i temps2,
            i temps2) == 1) goto end;

    for (i=0;i<16;i++)
        pat[3][counter[3]][1]=
        bitrep[0][bit[i][j][i][i]+
        bitrep[3][bit[3][j]][i];

}
else
{
    if (mulcrv(pgf1[0][i],pgf1[3][j],
            i temps,
            pgs1[0][i],pgs1[0][i],
            pgs1[3][j],pgs1[3][j],
            i temps2,
            i temps2) == 1) goto end;

    if (mulpoly(temp,pgf2[3][found],
            i temps2,
            i temps2,
            pgs2[3][found],
            pgs2[3][found],
            i temps2,
            i temps2) == 1) goto end;

    pgs2[3][found]=temps2;
    pgs2[3][found]=temps2;
    free(pgf2[3][found]);
    pgs2[3][found]=temps2;
    free(temp);
}

/* Find maximum power of two represented by product terms.*/

mpow=0;
    for (s=0;s<4;s++)
        for (i=0;i<counter[s];i++)
            for (j=0;j<16;j++)
                if (pat[s][i][j]*(j+1)>mpow)
                    mpow=pat[s][i][j]*(j+1);

/* Find maximum exponent of a variable in product terms, and maximum degrees.*/

mexp=0;
for (i=0;i<16;i++)
    maxdegree[i]=0;

for (s=0;s<4;s++)
    for (i=0;i<counter[s];i++)
        for (j=0;j<16;j++)
            [if (pat[s][i][j]>mexp)
                mexp=pat[s][i][j];
            if (maxdegree[j]<pat[s][i][j])
                maxdegree[j]=pat[s][i][j];

       /* Free memory used by pgfs of input polynomials, as they are no longer needed.*/

for (s=0;s<4;s++)
    for (i=0;i<count[s];i++)
        [free(pgfl[s][i]);

    /*Modify the product pgfs to take into account probabilities of negative values.*/
    prodpos[0]=pos[0]*pos[2]*neg[0]*neg[2];
    prodpos[1]=pos[1]*pos[3]*neg[1]*neg[3];
    prodpos[3]=pos[0]*pos[3]*neg[0]*neg[3];
    for (s=0;s<4;s++)
        for (i=0;i<counter[s];i++)
            redistr(&pgf2[s][i], prodpos[s].1-prod-
                    pos[s].&pgs2[s][i].pge2[s][i]); MARJ *

    /* This section of code calculates the pgfs of ac-bd and ad-bc. Since no new
       terms will be created, references
       are made to the list of terms in PAT. Care is taken since ad and bc, for
       instance, may not have all the same terms.*/

    /*Set up a new array, PAT2. The first index is 0 for real terms.
       The second is a term number. If the third index is 0, the value in the array is
       the
       term number in PAT's list of terms in ac. If it is 1, the value in the array is
       the
       term number in PAT's list of terms in bd. If the term does not appear in one of
       these lists
       the corresponding array value is -1*/

    /* Loop through the list of terms in ac in PAT. Add the ith term in the list to
       PAT2[0]. -1's are used
       to indicate that these terms have not yet been found in bd.*/

        for (i=0;i<counter[0];i++)
            [
pat2[0][i][0]=i;
pat2[0][i][1]=-1;
}

/*PAT2 now has the same number of terms as ac*/
counter2[0]=counter[0];

/* Loop through the list of terms in bd.*/
for (i=0;i<counter[1];i++)
{
    found=-1;
    for (j=0;j<counter[0];j++) /* search through the terms in ac for a
match.*/
        if (pat1[i][0]==pat0[j][0] &&
            pat1[i][1]==pat0[j][1] &&
            pat1[i][2]==pat0[j][2] &&
            pat1[i][3]==pat0[j][3] &&
            pat1[i][4]==pat0[j][4] &&
            pat1[i][5]==pat0[j][5] &&
            pat1[i][6]==pat0[j][6] &&
            pat1[i][7]==pat0[j][7] &&
            pat1[i][8]==pat0[j][8] &&
            pat1[i][9]==pat0[j][9] &&
            pat1[i][10]==pat0[j][10] &&
            pat1[i][11]==pat0[j][11] &&
            pat1[i][12]==pat0[j][12] &&
            pat1[i][13]==pat0[j][13] &&
            pat1[i][14]==pat0[j][14] &&
            pat1[i][15]==pat0[j][15])
            found=j; /* If this term appears in the list
for ac*/
    pat2[0][found][1]=i; /* insert the proper index in PAT2's list*/
    else
        { pat2[0][counter2[0]][0]=-1; /* Otherwise add the term to PAT2's
list.*/
            pat2[0][counter2[0]][1]=i;
            counter2[0]=counter2[0]+1;
        }
}

/* Loop through the terms in PAT2 */
for(i=0;i<counter2[0];i++)
{
    if (pat2[0][i][0]==-1) /* If the term does not exist in ac, then the pgf
for that term in */
        { /* ac-bd is the pgf of that term in
bd with some modifications.*/
            invert(&pgf3[0][i], pgf2[1][pat2[0][i][1]],
                &pgs3[0][i],&pgs3[0][i],
                pgs2[1][pat2[0][i][1]],
                pge2[1][pat2[0][i][1]]);
        }
    else if (pat2[0][i][1]==-1) /*If the term does not exist in bd, then the pgf for that term in*/
the random variable representing

pgf3[0][i]=pgf2[0][pat2[0][i][i][0]]; /* the coeff of that
term in ac */
pgs2[0][i]=pgs2[0][pat2[0][i][i][0]];
pge2[0][i]=pge2[0][pat2[0][i][i][0]];
}

else
{
if (multiply(pgfs[2][0][pat2[0][i][i][0]]). /* Otherwise, the pgf
of the term in ac-bd is the pgf of

pgf2[i][pat2[0][i][1][1]], /* the difference of the
random variables represented

pgs3[0][i], /* by the pgfs of the coeff

pgs2[0][pat2[0][i][i][0]],
pge2[0][pat2[0][i][i][0]],
pge2[1][pat2[0][i][1][1]],
pge2[1][pat2[0][i][1][1]],
&pgs3[0][i],
&pgs3[0][i]==1) goto end;

free(pgfs[2][0][pat2[0][i][i][0]]); /* The unneeded pgfs are dis-
carded to save memory. */
free(pgfs[2][1][pat2[0][i][1][1]]);
}

* Same as above but for the imaginary terms. Note that no subtraction of random
variables occurs in this case. */
for (i=0; i<counter[2]; i++)
{
pat2[1][i][0]=i;
pate[1][i][1]=-1;
}

counter2[1]=counter[2];

for (i=0; i<counter[1]; i++)
{
found=-1;
for (j=0; j<counter[2]; j++)
if (pat3[1][i][0]==pat2[2][j][0] &&
pat3[1][i][1]==pat2[2][j][1] &&
pat3[1][i][2]==pat2[2][j][2] &&
pat3[1][i][3]==pat2[2][j][3] &&
pat3[1][i][4]==pat2[2][j][4] &&
pat3[1][i][5]==pat2[2][j][5] &&
pat3[1][i][6]==pat2[2][j][6] &&
pat3[1][i][7]==pat2[2][j][7] &&
pat3[1][i][8]==pat2[2][j][8] &&
pat3[1][i][9]==pat2[2][j][9] &&
pat3[1][i][10]==pat2[2][j][10] &&
pat3[1][i][11]==pat2[2][j][11] &&
pat3[1][i][12]==pat2[2][j][12] &&
pat3[1][i][13]==pat2[2][j][13] &&
pat[3][i][14] = pat[2][j][14] &
pat[3][i][15] = pat[2][j][15])
    found = j;

if (found != -1)
    pat2[i][found][1] = i;
else
    {
        pat2[i][counter2[1]][0] = -1;
        pat2[i][counter2[1]][1] = i;
    }
}

for (i = 0; i < counter2[1]; i++)
{
    if (pat2[1][i][0] == -1)
        {
            pgf3[1][i] = pgf2[3][pat2[1][i][1]];  
            pgs3[1][i] = pgs2[3][pat2[1][i][1]];  
            pge3[1][i] = pge2[3][pat2[1][i][1]];  
        }
    else
        if (pat2[1][i][1] == -1)
            {
                pgf3[1][i] = pgf2[2][pat2[1][i][0]];  
                pgs3[1][i] = pgs2[2][pat2[1][i][0]];  
                pge3[1][i] = pge2[2][pat2[1][i][0]];  
            }
        else
            {
                if (multiply[pgf2[2][pat2[1][i][0]],
                       pgf2[3][pat2[1][i][1]],
                       &pgf3[1][i],
                       pgs2[2][pat2[1][i][0]],
                       pge2[2][pat2[1][i][0]],
                       &pgs3[1][i],
                       &pge3[1][i] == 1) goto end;
                free(pg2[2][pat2[1][i][0]]);
                free(pg2[3][pat2[1][i][1]]);
            }
}

/* Loop through all terms in ac-bd and order them in order of decreasing power of two using a bubble sort.*/
swap = 1;
while (swap == 1)
{
    swap = 0;
    for (i = 0; i < counter2[0] - 1; i++)
    {
        a = 0;
        b = 0;
        if (pat2[0][i][0] == -1)
c1=1;
else
  c1=0;
if (pat2[0][i+1][0]==-1)
  c2=1;
else
  c2=0;

a=pat[c1][pat2[0][i][c1]][0]+/* Find powers of two*/
  pat[c1][pat2[0][i][c1]][1]*2+
  pat[c1][pat2[0][i][c1]][2]*3+
  pat[c1][pat2[0][i][c1]][3]*4+
  pat[c1][pat2[0][i][c1]][4]*5+
  pat[c1][pat2[0][i][c1]][5]*6+
  pat[c1][pat2[0][i][c1]][6]*7+
  pat[c1][pat2[0][i][c1]][7]*8+
  pat[c1][pat2[0][i][c1]][8]*9+
  pat[c1][pat2[0][i][c1]][9]*10+
  pat[c1][pat2[0][i][c1]][10]*11+
  pat[c1][pat2[0][i][c1]][11]*12+
  pat[c1][pat2[0][i][c1]][12]*13+
  pat[c1][pat2[0][i][c1]][13]*14+
  pat[c1][pat2[0][i][c1]][14]*15+
  pat[c1][pat2[0][i][c1]][15]*16;

b=pat[c2][pat2[0][i+1][c2]][0]+/* Find powers of two*/
  pat[c2][pat2[0][i+1][c2]][1]*2+
  pat[c2][pat2[0][i+1][c2]][2]*3+
  pat[c2][pat2[0][i+1][c2]][3]*4+
  pat[c2][pat2[0][i+1][c2]][4]*5+
  pat[c2][pat2[0][i+1][c2]][5]*6+
  pat[c2][pat2[0][i+1][c2]][6]*7+
  pat[c2][pat2[0][i+1][c2]][7]*8+
  pat[c2][pat2[0][i+1][c2]][8]*9+
  pat[c2][pat2[0][i+1][c2]][9]*10+
  pat[c2][pat2[0][i+1][c2]][10]*11+
  pat[c2][pat2[0][i+1][c2]][11]*12+
  pat[c2][pat2[0][i+1][c2]][12]*13+
  pat[c2][pat2[0][i+1][c2]][13]*14+
  pat[c2][pat2[0][i+1][c2]][14]*15+
  pat[c2][pat2[0][i+1][c2]][15]*16;

if (b>a) /* If terms out of order, swap
them.*/
{
  t=pat2[0][i][0];
  pat2[0][i][0]=pat2[0][i+1][0];
  pat2[0][i+1][0]=t;
  t=pat2[0][i+1][1];
  pat2[0][i+1][1]=pat2[0][i+1][1];
  pat2[0][i+1][1]=t;
  swap=1;
  temp=pgf3[0][i];
  pgf3[0][i]=pgf3[0][i+1];
  pgf3[0][i+1]=temp;
}
ps=pgs3[0][i];
pgs3[0][i]=pgs3[0][i+1];
pgs3[0][i+1]=ps;

pe=pge3[0][i];
pge3[0][i]=pge3[0][i+1];
pge3[0][i+1]=pe;

}
}

swap=i;
while (swap==1)
{

/* Same as above but for ad+bc*/
swap=0;
for (i=0;i<counter2[1]-1;i++)
{
  a=0;
  b=0;
  if (pat2[1][i][0]==-1)
    c1=1;
  else
    c1=0;
  if (pat2[1][i+1][0]==-1)
    c2=1;
  else
    c2=0;
  a=pat[c1+2][pat2[1][i][c1]][0]*
    pat[c1+2][pat2[1][i][c1]][1]*2-
    pat[c1+2][pat2[1][i][c1]][2]*3-
    pat[c1+2][pat2[1][i][c1]][3]*4-
    pat[c1+2][pat2[1][i][c1]][4]*5-
    pat[c1+2][pat2[1][i][c1]][5]*6-
    pat[c1+2][pat2[1][i][c1]][6]*7-
    pat[c1+2][pat2[1][i][c1]][7]*8-
    pat[c1+2][pat2[1][i][c1]][8]*9-
    pat[c1+2][pat2[1][i][c1]][9]*10-
    pat[c1+2][pat2[1][i][c1]][10]*11-
    pat[c1+2][pat2[1][i][c1]][11]*12-
    pat[c1+2][pat2[1][i][c1]][12]*13-
    pat[c1+2][pat2[1][i][c1]][13]*14-
    pat[c1+2][pat2[1][i][c1]][14]*15-
    pat[c1+2][pat2[1][i][c1]][15]*16;

  b=pat[c2+2][pat2[1][i+1][c2]][0]*
    pat[c2+2][pat2[1][i+1][c2]][1]*2-
    pat[c2+2][pat2[1][i+1][c2]][2]*3-
    pat[c2+2][pat2[1][i+1][c2]][3]*4-
    pat[c2+2][pat2[1][i+1][c2]][4]*5-
    pat[c2+2][pat2[1][i+1][c2]][5]*6-
    pat[c2+2][pat2[1][i+1][c2]][6]*7-
    pat[c2+2][pat2[1][i+1][c2]][7]*8-
    pat[c2+2][pat2[1][i+1][c2]][8]*9+
if (b>a)
  
  t=pat2[1][i][0];
  pat2[1][i][0]=pat2[1][i-1][0];
  pat2[1][i-1][0]=t;
  t=pat2[1][i][1];
  pat2[1][i][1]=pat2[1][i-1][1];
  pat2[1][i-1][1]=t;
  swap=1;

  temp=pgf3[1][i];
  pgf3[1][i]=pgf3[1][i-1];
  pgf3[1][i-1]=temp;

  ps=pgs3[1][i];
  pgs3[1][i]=pgs3[1][i-1];
  pgs3[1][i-1]=ps;

  pe=pge3[1][i];
  pge3[1][i]=pge3[1][i-1];
  pge3[1][i-1]=pe;

} */

"* Open the probability file.*/
strcpy(tname,"temp.pgf");
hfo=fopen("temp.pgf","wb");
if (hfo==NULL)
{
  printf("Temporary file could not be created!");
  goto end;
}

"* Find maximum bit in the block length.*/
l=maxb(bl,bl);

"* Loop through the real terms.*/
for(s=0; s<counter2[0]; s++)
{
  if (pat2[0][s][0]==-1)
    k=1;
  else
k=0;

    /* Write their flag representations to the probability file, followed by
     a 0 indicating real terms. */
    for (i=0;i<16;i++)
    {
        fwrite(&pat[k][pat2[0][s][k]][i].sizeof(int),1,hfo);
    }

    m=0;
    fwrite(&m,sizeof(int),1,hfo);

    /* The pgf of (block length) rand. vars representing the coeff's added
     together is calculated in the following manner:

     Note that squaring a pgf, we obtain the pgf of a random variable added to
     itself.

     Let PGF be the eventual output coeff. Let CURRENT initially be the pgf of
     the term's coefficient.
     To obtain the pgf of the sum of 2^i random variables representing the
     coeff, square CURRENT i-1 times.

     If the least bit of the block length is 0, the PGF will initially be "1".
     Otherwise, it will be set to CURRENT, and will hence represent the pgf of
     1 of the random variables
     being treated.

     Square CURRENT so that it represents the pgf of the sum of two random
     variables.
     If the second least bit of the block length if 1, multiply PGF by CURRENT,
     thereby computing the
     pgf of the sum of however many random variables PGF represented, plus two
     more.

     Continue in this manner, squaring CURRENT and multiplying it with PGF
     whenever the current bit of the
     block length is 1, to obtain the final output pgf. */

    /* Initialize PGF */
    for(i=0;i<5; i++)
    {
        pgf=pgf3[0][i];
        ps=pgs3[0][i];
        pe=pge3[0][i]; /* MRBJ */
        printf("ps=%i \n pe=%i \n", ps, pe);
        for(j=0;j<=pe-ps;j++)
            printf("%2.15lf \n", pgf[j]);
    }
    if (bl%2==1)
    {
        pgf=pgf3[0][s];
        ps=pgs3[0][s];
        pe=pge3[0][s];
        poffset=0;
    }
/* Add the offset to the least significant exponent of the output Pgf. */

/* If the current bit of the block length is 1, multiply Pgf and current, freeing the unsquared Pgf if possible. */

/* Loop through all bits of the block length. */

for (i = 0; i < b; i++)
{
  if (current & (1 << i))
  {
    free(current);
    current = pgf; /* Pgf gets squared. */
  }
  else
  {
    if (current == NULL)
    {
      pgf = current;
      current = NULL;
      /* Current gets cleared. */
      goto end;
    }
    else
    {
      if (current->working)
      {
        /* Work current, freeing the unsquared Pgf if possible. */
        if (current->working
          for (i = 0; i < b; i++)
          
        /* Square current, freeing the unsquared Pgf if possible. */
        if (current->working
          current = pgf; /* Pgf gets squared. */

        /* Current gets cleared. */
        goto end;
      }
      else
      {
        pgf = current->pgf;
        current = NULL;
        /* Current gets cleared. */
        goto end;
      }
    }
  }
}

end:
ps=ps+poffset;

    /* write the least and greatest significant exponents of the output pgf to the probability file.*/
    fwrite(&ps,sizeof(int),1,hfo);
    fwrite(&pe,sizeof(int),1,hfo);

    /* Write all significant coefficients of the output pgf to the probability file.*/
    for (i=ps;i<=pe;i++)
        fwrite(&pgf[i-ps+poffset],sizeof(double),1,hfo);

    /* Free the now unneeded pgfs for this term.*/
    if (pgf!=current)
        free(current);
    free(pgf);

    /* Same as above, but for imaginary terms. The only difference is that a 1 is written after the
     * term representation instead of a 0, to indicate an imaginary term.*/
    for(s=0;s<counter2[1];s++)
        {
            if (pat2[1][s][1]==1)
                k=0;
            else
                k=1;
            for (i=0;i<16;i++)
                {
                    fwrite(&pat[k][pat2[1][s][k]][i],sizeof(int),1,hfo);
                }
            m=1;
            fwrite(&m,sizeof(int),1,hfo);

            if (bl%2==1)
                {
                    pgf=pgf3[1][s];
                    ps=pgs3[1][s];
                    pe=pgel[1][s];
                    poffset=0;
                }
            else
                pgf=NULL;

            current=pgf3[1][s];
            cus=pgs3[1][s];
            cue=pgel[1][s];
            cuoffset=0;
            for (i=1;i<=1;i++)
                {
\{ 
if (multsqr(current, &working, 
cus, cue, &ws, &we, 
cuoffset, &woffset) == 1) goto end; 
if (current != pgf) 
\{ 
  free(current); 
\} 
current = working; 
cus = ws; 
cue = we; 
cuoffset = woffset; 
if ((bl / (int)powr(2, i)) % 2 == 1) 
\{ 
  if (pgf == NULL) 
  \{ 
    pgf = current; 
    ps = cus; 
    pe = cue; 
    poffset = cuoffset; 
  \} 
  else 
  \{ 
    if (multpolh(current, pgf, 
                  &working, 
                  cus, cue, ps, pe, 
                  &ws, &we, 
                  cuoffset, poffset, 
                  &woffset) == 1) 
      goto end; 
      free(pgf); 
      pgf = working; 
      ps = ws; 
      pe = we; 
      poffset = woffset; 
  \} 
\} 
ps = ps + poffset; 
fwrite(&ps, sizeof(long), 1, hfo); 
fwrite(&pe, sizeof(long), 1, hfo); 
for (i = ps; i <= pe; i++) 
  fwrite(&pgf[i - ps + poffset], sizeof(double), 1, hfo); 
if (pgf != current) 
\{ 
  free(pgf); 
\} 
free(current); 
\}

/* Terminate the probability file by a marker string of 999's*/

i = -999; 
fwrite(&i, sizeof(int), 1, hfo); 

i=999;
fwrite(&i,sizeof(int),1,hfo);
i=999;
fwrite(&i,sizeof(int),1,hfo);

/* Close the output file, activate the appropriate buttons, and set ccheck to indicate
that calculations are complete. */
close(hfo);

ccheck=1;

return 0;

/* This section of code is only executed if an error occurred in the calculations due to
memory or disk problems. The only difference is that ccheck is not set. */
end: ccheck=0;

return 1;
}

/* Procedure that saves a model to disk. */

int save()
{
    FILE *f;
    int i,j,k,answer;
    char *p,"q; char pname[81];
    char dname[81]; char qname[81];
    char xname[81];

    /* Strip the extension from the filename*/
    q=NULL;
    for (p=sname; p!=NULL;p++)
        if ("p=",')
            q=p;

    /* Generate the filename and probability filename */
    if (q!=NULL)
    {
        strcpy(pname,sname);
        strcpy(qname,sname);
        strcat(pname,".pgf");
        strcat(sname,".mod");
    }
    else
    {
        *q=NULL;
        strcpy(pname,sname);
        strcpy(qname,sname);
        strcat(pname,".pgf");
        strcat(sname,".mod");
    }
/* If calculations have been completed, rename the filename of the probability file to its new name. */
if (ccheck==1)
{
    f=fopen(pname,"r");

    /* If the probability file is already saved under its new name, nothing needs to be done. */
    if (strcmp(tname,pname)==0)
        goto sav;

    /* Open the model file */
    sav:f=fopen(sname,"wb");
    if (f==NULL)
    {
        printf("Error! File could not be created.");
        return 1;
    }

    /* Write all model specifications to this file. */
    for (i=0;i<4;i++)
    {
        for (j=0;j<15;j++)
            fwrite(&v[i][j],sizeof(int),1,f);
        fwrite(&vcheck[i],sizeof(int),1,f);
        fwrite(&dcheck[i],sizeof(int),1,f);
        fwrite(&chcheck[i],sizeof(int),1,f);
        fwrite(&lcheck[i],sizeof(int),1,f);
        fwrite(&avval[i],sizeof(int),1,f);
        fwrite(&varval[i],sizeof(int),1,f);
        fwrite(&uvar[i],sizeof(int),1,f);
        fwrite(&sumval[i],sizeof(int),1,f);
        fwrite(&sumaxval[i],sizeof(int),1,f);
        fwrite(&astartval[i],sizeof(int),1,f);
        fwrite(&srendval[i],sizeof(int),1,f);
        fwrite(&amaxbit[i],sizeof(int),1,f);
        fwrite(&kbcheck[i],sizeof(int),1,f);
        for (j=0;j<11;j++)
        {
            for (k=0;k<16;k++)
                fwrite(&bitrep[i][j][k],sizeof(int),1,f);
            fwrite(&brep[i][j],sizeof(int),1,f);
            fwrite(&bitreps[i][j][0],199.1,f);
        }
    fwrite(&bl,sizeof(int),1,f);
    fwrite(&scheck,sizeof(int),1,f);
    fwrite(&name[0],50,1,f);
    fwrite(&pname[0],50,1,f);
    fwrite(&mexp,sizeof(int),1,f);
    fwrite(&mpow,sizeof(int),1,f);
    fclose(f);
    saved=1;
    return 0;
}
/* This procedure is called when the OK button in the Save window is pressed. */

int saveokproc ()
{
    char xname[80], *p, *q;
    FILE *iFile;
    int answer;

    /* Get the filename from the window, strip the extension and replace it with .mod */
    printf("nEnter file name: ");
    fflush(stdout);
    scanf("%s", xname);

    q=NULL;
    for (p=xname; p!=NULL; p++)
        if (*p=='.')
            q=p;

    if (q==NULL)
        { 
        strcat(xname, ".*mod");
        }
    else
        { 
        "q=NULL;
        strcat(xname, ".*mod");
        }

    /* Check to see that the given filename is valid. */
    iFile=fopen(xname "w");
    if (iFile==NULL)
        { 
        printf("File could not be opened! ");
        return 1;
        }
    fclose(iFile);

    /* Save the model */
    strcpy(sname, xname);
    if (save==1)
        return;

    named=1;
}

int loadokproc (item, event)
{
    Panel_item item;
    Event event;
    {
char xname[80], *p, *q;
FILE *iFile;
int answer;

/* Get the filename from the window, strip the extension, and replace it with .mod */
strcpy(xname, xv_get(loadtext, PANEL_VALUE));
q=NULL;
for (p=xname; *p!=NULL; p++)
    if (*p=='.')
        q=p;
if (q==NULL)
    {
        strcat(xname, "*.mod");
    }
else
    {
        "q=NULL;
        strcat(xname, "*.mod");
    }

/* Check to make sure that the file exists.*/
iFile=fopen(xname, "r");
if (iFile==NULL)
    {
        notice_prompt(savepanel, NULL,
            NOTICE_MESSAGE_STRINGS,
            "File could not be opened!", NULL,
            NOTICE_BUTTON."OK", 100,
            NULL);
        return 1;
    }
fclose(iFile);

/* If it does exist, load the model*/
strcpy(sname, xname);
if (load()==1)
    {
        return;
    }

/* If successful activate buttons, close window, and reactivate File button.*/
activate();
xv_set(loadframe, XV_SHOW, FALSE, NULL);
xv_set(filebutton, PANEL_INACTIVE, FALSE, NULL);
}

/* This procedure, given the product of moduli being used, returns the maximum error
that would occur, and what term it would occur in.*/

int GetWorst()
{  
    double terr; /* Error accumulator*/  
    int i,t,j; /* Counters and dummy variables.*/  
    long e,s; /* Start and end values of distributions*/  
    double temp; /* Data is read into this variable from the probability file.*/  
    long tcurr; /* Value of current offset into file.*/  
    int exp,exptemp; /* exp holds the power of two represented by the term. exptemp is used to calc. exp.*/  
    int twopwr; /* used to calculate exp */  
    merr=0; /* Initialize maximum error to 0 */  

    fseek(f,0,0); /* Goto the start of the file. */  
    tcurr=0; /* Set current file position.*/  
    indx[0]=0; /* Set position of first term in file. */  
    j=0; /* Set count of current term in file.*/  

    fread(&t,sizeof(int),1,f); /* Read value from file to see that the end has not been reached.*/  
    exp=t; /* Copy this value into exp, since it is required in computing power of two.*/  

    while(t!=999) /* While the end has not been reached...*/  
    {  
        /* Compute power of two represented by current term.*/  
        twopwr=2;  
        for (i=1;i<16;i++)  
        {  
            fread(&exptemp,sizeof(int),1,f);  
            exp=exp*exptemp*twopwr;  
            twopwr=twopwr*2;  
        }  

        fread(&i,sizeof(int),1,f); /* Read dummy value.*/  
        terr=0; /* Initialize error accumulator*/  
        fread(&s,sizeof(long),1,f); /* Read in start and end values of distribution*/  

        /* If there are negative values in the distribution, add to terr the probabilities of the values less than -1*(mmod+1)/2*/  

        if (s<0)  
            for (i=s;i<=-1*(mmod+1)/2;i++)  
            {  
                fread(&temp,sizeof(double),1,f);  
                terr=terr+temp;  
            }  

        /* If there are values in the distribution greater than (mmod+1/2), add their probs. to terr.*/  
    }  
}
if (e>(mm+1)/2)
{
    fseek(f, tcurr+17*sizeof(int)+2*sizeof(long)+
        ((mm+1)/2-s)*sizeof(double), 0);

    for (i=(mm+1)/2; i<=e; i++)
    {
        fread(&temp, sizeof(double), 1, f);
        tcurr=temp;
    }
}

    /* If the error accumulated in tcurr is greater than the maximum
       error...*/
    if (tcurr>merr && exp>lsb)
    {
        wj=j;   /* Worst term # is current term.*/
        merr=tcurr; /* Max. error is tcurr.*/
        tcurr=tcurr; /* Position of worst term in file is current position.*/

        fseek(f, tcurr+17*sizeof(int)-2*sizeof(long)+
            (e-s-1)*sizeof(double), 0);

        fread(&t, sizeof(int), 1, f); /* Read value to check for end of file.*/
        exp=t;
        tcurr=tcurr+17*sizeof(int)-2*sizeof(long)+(e-s-1)*sizeof(double); /* Set new current position in file.*/
        j=j+1; /* Update count of terms.*/
        indx[j]=tcurr; /* Update index array.*/
    }

    indx[j]=-1; /* Set final index to -1 to indicate the end of terms.*/
    fseek(f, curr, 0); /* Seek to the position of the worst case term.*/
    wcurr=curr; /* Set the position of the worst case term*/
cj=wj;   /* Set the index of the worst case term.*/
    return 0;
}

/* This procedure, given the maximum error, returns the minimum necessary product
   of moduli required, and the term the maximum error occurs in.*/

int GetWorstP()
{
    double tcurr; /* Error accumulator.*/
    int i, t, j;  /* Counters and temporary variables.*/
    long e, s; /* Start and end values of coefficient of term.*/
    long w1, w2; /* Current high and low values of coefficient being processed.*/
    long tcurr; /* Current position in the probability file.*/
int exp, exptemp; /* Exptemp is used to calc. the power of 2 represented by a term. Value is stored in exp. */
int twopwr;  /* Used to calculate exp. */
int tmod;  /* Temporary product of moduli. */
double *temp;  /* Pointer to data read in from file. */
double *temp2; /* Pointer offset from temp to simplify indexing. */

mmod=0;  /* Initialize maximum product of moduli. */

fseek(f, 0, 0);  /* Seek to start of file. */
tcurr=0;

j=0;  /* Holds number of current term. */
indx[0]=0;  /* Holds position of current term. */

fread(&t, sizeof(int), 1, f); /* Read value to check for EOF. Value maintained in exp since used later on. */
exp=t;

while(t!=-999) /* While NOT EOF... */
{
  /* Calculate power of two represented by term. */
  twopwr=2;
  for (i=1; i<16; i++)
  {
    fread(&exptemp, sizeof(int), 1, f);
    exp=exp-exptemp*twopwr;
    twopwr=twopwr+1;
  }

  /* Skip complex/real flag */
  fread(&i, sizeof(int), 1, f);

  /* Initialize temporary error. */
  terr=0;

  /* Read start and end values of coefficient. */
  fread(&s, sizeof(long), 1, f);
  fread(&e, sizeof(long), 1, f);

  /* Allocate memory to hold pgf of coefficient. */
  temp=(double *)malloc((e-s+1)*sizeof(double));

  if (temp==NULL)
  {
    printf("Insufficient Memory! ");
    return 1;
  }

  /* 0 coeff of pgf will be at temp2[0]. */
  temp2=(double *) temp-s;

  /* Set current low and high values of coefficient to coefficient’s extreme values. */
  if (e>0)
  w2=e;
else
    w2=0;
if (s<0)
    w1=s;
else
    w1=0;

  /* Read pgf into array*/
  for (i=s;i<=e;i++)
    fread(&temp2[i].sizeof(double),1,f);

  /* While the temporary error is less than the specified error, add to the
   error accumulator the
   probability that the coefficient will take on the value of its current
   high and/or low value,
   depending on which of these values is greatest in absolute value. Adjust
   the values of the current
   high and low values so that the same error will not be added twice to the
   accumulator.*/

  while(terr<merr)
    if (abs(w1)==w2)
      {
        terr=terr+temp2[w1]+temp2[w2];
        w1=w1+1;
        w2=w2+1;
      }
    else
      if (abs(w1) >w2)
        {
          terr=terr+temp2[w1];
          w1=w1+1;
        }
      else
        {
          terr=terr+temp2[w2];
          w2=w2+1;
        }
  
  /* Free the pgf array, since it is no longer needed.*/
  free(temp);

  /* From the values of the current high and low values, calculate a mini-
   mum product of moduli that would generate
   the required error.*/
  if (abs(w1)>abs(w2))
    tmod=(abs(w1)+1.5)*2;
  else
    tmod=(abs(w2)+1.5)*2;

  /* Correction factor for when the desired error is 0%*/
  if (w1==s & & w2==e)
    tmod=tmod-2;

  /* If the product of moduli just calculated is the largest yet for any
   term, let this be the current maximum
   product of moduli, and let the current term be the worst case term.*/
if (tmod%mmmod & exp>=lsb)
{
    wj=j; /* # of worst term is # of current term.*/
    mmod=tmod;/* Maximal product of moduli is current product of mod-
    uli.*/
    curr=tcurr;/* Offset of worst term in probability file is current
    offset.*/
}

fseek(f,tc curr+17*sizeof(int)+2*sizeof(long)+/* Seek to the next term in
the file.*/
    (e-s+1)*sizeof(double),0);

fread(&t,sizeof(int),1,f);" /* Read a value as before to check for EOF.*/
exp=t;

/* Update current offset in file, current term #, and add new term offset
to indx array.*/
tcurr=tc curr+17*sizeof(int)+2*sizeof(long)+(e-s+1)*sizeof(double);
    j=j-1;
    indx[j]=tcurr;
}

indx[j]=-1; /* Indicates that there are no more terms.*/

fseek(f,curr,0);" /* Seek to position of worst case term.*/
wc curr=curr; /* Set offset of worst case term to offset of current term.*/
    cj=wj; /* Set term # of worst case term to # of worst case
term.*/

return 0;
}

/* This procedure is called to calculate the error of the current term, and display
this error, as well as a symbolic representation
of the term.*/

int geterr()
{
    int i,i; /* Counters.*/
    char estr[20];" /* Temporary string.*/
    char bstr[200],tempstr[20];" /* Temporary strings used to generate symbolic rep-
    resentation.*/
    int t; /* Temporary variable.*/
    char *p; /* Temporary pointer used to trim trailing * from symbolic representa-
tion.*/
    double terr;" /* Temporary error.*/
    double temp;" /* Holds values read in from probability file.*/
    long e,s,tc curr;" /* Start and end values of coefficient of current term, and off-
    set of current term, respectively.*/

    fseek(f,curr,0);" /* Seek to the data for the current term.*/
    strcpy(bstr," ");" /* Initialize symbolic representation of term.*/

    fread(&t,sizeof(int),1,f);" /* If the variable 2 has non-zero power in the term,
    concat. 2^(power) to */
    if (t>0) /* symbolic representation.*/
    {
    
}
```c
    sprintf(tempstr,"%i",t);
    strcat(bstr,"2" );
    strcat(bstr,tempstr);
    strcat(bstr,"**");
}

fread(&t,sizeof(int),1,f); /* Same as above, but for variable 4, etc. */
if (t>0)
    {
        sprintf(tempstr,"%i",t);
        strcat(bstr,"4" );
        strcat(bstr,tempstr);
        strcat(bstr,"**");
    }

fread(&t,sizeof(int),1,f);
if (t>0)
    {
        sprintf(tempstr,"%i",t);
        strcat(bstr,"8" );
        strcat(bstr,tempstr);
        strcat(bstr,"**");
    }

fread(&t,sizeof(int),1,f);
if (t>0)
    {
        sprintf(tempstr,"%i",t);
        strcat(bstr,"16" );
        strcat(bstr,tempstr);
        strcat(bstr,"**");
    }

fread(&t,sizeof(int),1,f);
if (t>0)
    {
        sprintf(tempstr,"%i",t);
        strcat(bstr,"32" );
        strcat(bstr,tempstr);
        strcat(bstr,"**");
    }

fread(&t,sizeof(int),1,f);
if (t>0)
    {
        sprintf(tempstr,"%i",t);
        strcat(bstr,"64" );
        strcat(bstr,tempstr);
        strcat(bstr,"**");
    }

fread(&t,sizeof(int),1,f);
if (t>0)
    {
        sprintf(tempstr,"%i",t);
        strcat(bstr,"128" );
        strcat(bstr,tempstr);
        strcat(bstr,"**");
    }
```
fread(&t, sizeof(int), 1, f);
if (t>0)
{
    printf(tempstr, "%i", t);
    strcat(bstr, "256\n");
    strcat(bstr, tempstr);
    strcat(bstr, "\n");
}

fread(&t, sizeof(int), 1, f);
if (t>0)
{
    printf(tempstr, "%i", t);
    strcat(bstr, "512\n");
    strcat(bstr, tempstr);
    strcat(bstr, "\n");
}

fread(&t, sizeof(int), 1, f);
if (t>0)
{
    printf(tempstr, "%i", t);
    strcat(bstr, "1024\n");
    strcat(bstr, tempstr);
    strcat(bstr, "\n");
}

fread(&t, sizeof(int), 1, f);
if (t>0)
{
    printf(tempstr, "%i", t);
    strcat(bstr, "2048\n");
    strcat(bstr, tempstr);
    strcat(bstr, "\n");
}

fread(&t, sizeof(int), 1, f);
if (t>0)
{
    printf(tempstr, "%i", t);
    strcat(bstr, "4096\n");
    strcat(bstr, tempstr);
    strcat(bstr, "\n");
}

fread(&t, sizeof(int), 1, f);
if (t>0)
{
    printf(tempstr, "%i", t);
    strcat(bstr, "8192\n");
    strcat(bstr, tempstr);
    strcat(bstr, "\n");
}

fread(&t, sizeof(int), 1, f);
if (t>0)
{
    printf(tempstr, "%i", t);
}
strcat(bstr, "16384");
strcat(bstr, tempstr);
strcat(bstr, "**");
}

fread(&t, sizeof(int), 1, f);
if (t>0)
{
    sprintf(tempstr, "%i", t);
    strcat(bstr, "32768");
    strcat(bstr, tempstr);
    strcat(bstr, "**");
}

fread(&t, sizeof(int), 1, f);
if (t>0)
{
    sprintf(tempstr, "%i", t);
    strcat(bstr, "65536");
    strcat(bstr, tempstr);
    strcat(bstr, "**");
}

if (strcmp(bstr, "") == 0) /* If the representation is still blank, it is 1, so let
    the representation be "2^0" */
    strcpy(bstr, "2^0");
else
{
    l=strlen(bstr); /* Otherwise, a trailing * must be deleted from the rep-
representation. */
    p=bstr+l-1;
    *p=0;
}

fread(&t, sizeof(int), 1, f); /* Read in the complex/real flag, and concatenate the
appropriate modifier to the*/
if (t==1) /* symbolic representation. */
    strcat(bstr, " (Imag.)*");
else
    strcat(bstr, " (Real)*");

printf("%s", bstr); /* Display the symbolic representation in the window. */

terr=0; /* Set the error accumulator to 0, then read in min
and max values of coeff. of term. */
fread(&s, sizeof(long), 1, f);
fread(&e, sizeof(long), 1, f);

/* If the minimum value of the term is less than -1* half the product of the
moduli, add to error accumulator
the probabilities of the coeff of the term having value less than -1 times half
the product of the moduli.*/
if (s<0)
    for (i=s; i<=-(mmod+1)/2; i++)
    {
        fread(&temp,sizeof(double),1,f);
        terr=terr+temp;
    }

    /* If the maximum value of the term is greater than half the product of the mod-
     * uli, add to error accumulator
     * the probabilities of the coeff of the term having value greater than half the
     * product of the moduli. */

    if (e>(mmod+1)/2)
    {
        fseek(f,curr+17*sizeof(int)+2*sizeof(long)+/* Seek to appropriate posi-

        (((mmod+1)/2)-s)*sizeof(double),0);

        for (i=(mmod+1)/2; i<=e; i++)
        {
            fread(&temp,sizeof(double),1,f);
            terr=terr+temp;
        }
    }

    /* Display the error calculated above in the window. */
    printf(estr,"%f",terr*100);
    printf("%f",estr);

    fseek(f,curr,0); /* Seek to the current position in the file. */

    return 0;

    */

/* Same as the previous procedure, except the error need not be calculated, since it is
available. This procedure is used
only when originally displaying the worst case term's representation and error. In
addition, this procedure copies the worst
case error into werrstr, and the worst case representation into wstr for later use. */

int DisplBit()
{
    char estr[20];
    char tempstr[20],bstr[200];
    int t,1;
    char *p;
    sprintf(estr,"%f",terr*100);
    printf("%f",estr);
    strcpy(werrstr,estr);

    strcpy(bstr, "");
    fread(&t,sizeof(int),1,f);
    if (t>0)
    {
        sprintf(tempstr,"%i",t);

strcat(bstr,"2^" );
strcat(bstr,tempstr);
strcat(bstr,"^" );
}
fread(&t,sizeof(int),1,f);
if (t>0)
{
    printf(tempstr,"%i",t);
    strcat(bstr,"4^" );
    strcat(bstr,tempstr);
    strcat(bstr,"^" );
}
fread(&t,sizeof(int),1,f);
if (t>0)
{
    printf(tempstr,"%i",t);
    strcat(bstr,"8^" );
    strcat(bstr,tempstr);
    strcat(bstr,"^" );
}
fread(&t,sizeof(int),1,f);
if (t>0)
{
    printf(tempstr,"%i",t);
    strcat(bstr,"16^" );
    strcat(bstr,tempstr);
    strcat(bstr,"^" );
}
fread(&t,sizeof(int),1,f);
if (t>0)
{
    printf(tempstr,"%i",t);
    strcat(bstr,"32^" );
    strcat(bstr,tempstr);
    strcat(bstr,"^" );
}
fread(&t,sizeof(int),1,f);
if (t>0)
{
    printf(tempstr,"%i",t);
    strcat(bstr,"64^" );
    strcat(bstr,tempstr);
    strcat(bstr,"^" );
}
fread(&t,sizeof(int),1,f);
if (t>0)
{
    printf(tempstr,"%i",t);
    strcat(bstr,"128^" );
    strcat(bstr,tempstr);
    strcat(bstr,"^" );
}
fread(&t, sizeof(int), 1, f);
if (t>0)
{
    sprintf(tempstr, "%i", t);
    strcat(bstr, "255\n");
    strcat(bstr, tempstr);
    strcat(bstr, \n");
}
fread(&t, sizeof(int), 1, f);
if (t>0)
{
    sprintf(tempstr, "%i", t);
    strcat(bstr, "512\n");
    strcat(bstr, tempstr);
    strcat(bstr, \n");
}
fread(&t, sizeof(int), 1, f);
if (t>0)
{
    sprintf(tempstr, "%i", t);
    strcat(bstr, "1024\n");
    strcat(bstr, tempstr);
    strcat(bstr, \n");
}
fread(&t, sizeof(int), 1, f);
if (t>0)
{
    sprintf(tempstr, "%i", t);
    strcat(bstr, "2048\n");
    strcat(bstr, tempstr);
    strcat(bstr, \n");
}
fread(&t, sizeof(int), 1, f);
if (t>0)
{
    sprintf(tempstr, "%i", t);
    strcat(bstr, "4096\n");
    strcat(bstr, tempstr);
    strcat(bstr, \n");
}
fread(&t, sizeof(int), 1, f);
if (t>0)
{
    sprintf(tempstr, "%i", t);
    strcat(bstr, "8192\n");
    strcat(bstr, tempstr);
    strcat(bstr, \n");
}
fread(&t, sizeof(int), 1, f);
if (t>0)
{
    sprintf(tempstr, "%i", t);
}
strcat(bstr,"16384"");
strcat(bstr,tempstr);
strcat(bstr,""");
}

fread(&t,sizeof(int),1,f);
if (t>0)
{
    fprintf(tempstr,"%i",t);
    strcat(bstr,"32768")
    strcat(bstr,tempstr);
    strcat(bstr,""");
}

fread(&t,sizeof(int),1,f);
if (t>0)
{
    fprintf(tempstr,"%i",t);
    strcat(bstr,"65536")
    strcat(bstr,tempstr);
    strcat(bstr,""");
}

if (strcmp(bstr,"")==0)
    strcpy(bstr,"2")
else
{
    l=strlen(bstr);
    p=bstr+1-l;
    *p=0;
}

fread(&t,sizeof(int),1,f);
if (t==1)
    strcat(bstr," (Imag.)")
else
    strcat(bstr," (Real)")

printf("%s",bstr);
strcpy(wbstr,bstr);
return 0;
}

/* Calculates the number of moduli required, given the product of the moduli, the type of moduli, and the size of moduli.*/

int findsize()
{
    int i,s;
    long temp;

    /* Set offset into moduli array to reflect the type of moduli being used and their size.*/
    s=size-3*mod;
/* Set the temporary product of moduli to 1 and initialize number of moduli required. */
    temp=1;
    i=0;

    /* While the factors of the moduli of the current type and size are greater than the maximum exponent of the output terms
     (To permit the existence of inverses for reverse mapping), and the end of the list of moduli has not been reached
     (there is a maximum of 15 moduli of any given type & size), and the temporary product of moduli is less than the
     required product of moduli.... */

    while (sfact(mod[s][i])>nexp && mod[s][i]!=-1 && i<15 && temp<mmmod )
    {
        /* Multiply the temporary product of moduli by the current moduli and
         increase the count of moduli required. */
        temp=temp*mod[s][i];
        i=i+1;
    }

    /* If there is no way to obtain the required product of moduli, set the required
    number of moduli to -1, else set it to the number calculated. */

    if (temp<mmmod)
        n=-1;
    else
        n=i;
    return 0;
}

/* Returns the smallest prime factor of the input number. */
int sfact(a)
    int a;
    {
        int i;
        for (i=2;i<=a;i++)
            if (a%i==0)
                return i;
    }

/* Given the number of moduli to be used, and their type, calculate the maximum size of moduli required, in bits. */
int findnum()
    {
        int i,s,found;
        long temp;

        s=mod;" Set offset into moduli array to the least size moduli of the current type.";
        found=0;" Set the flag indicating that the required product of moduli has been attained to 0.";
/* While there remain moduli of the current type, and the required product has not yet been attained. */
   while ((s<4+omod && ckt<2) || s<14+omod) && found==0)
   {
      /* Initialize the count of moduli used, and the temporary product of moduli. */
      i=0;
      temp=1;

      /* While the moduli have factors large enough to have inverses in the reverse mapping, and the end of the current list of moduli has not been reached, and the maximum number of moduli allowed has not been reached... */
      while (sfact(mod[s][i])>mexp && mod[s][i]!=-1 && i<n)
      {
         /* Multiply the current temporary product of moduli by the current moduli, and increase the count of moduli used. */
         temp=temp*mod[s][i];
         i=i+1;

         /* If the temporary product of moduli has equaled or exceeded the required product of moduli, but fewer than the allowable number of moduli were required, set found =2. */
         if (temp>mmod && i<n)
            found=2;

         /* If the temporary product of moduli has equaled or exceeded the required product of moduli, and exactly the allowable number of moduli have been used, set found =1. */
         if (temp == mmod && i==n)
            found=1;
      }
      /* Increment the pointer to the next list of moduli. */
      s=s+1;
   }

   /* If found=1, return the calculated size of moduli required (adjusted since an array offset was actually calculated. */
   if (found==1)
      size=s+2-omod;

   /* If the desired product of moduli could not be obtained, return -1. */
   if (found==0)
      size=-1;

   /* If found =2, return -1 times the calculated size of moduli required (adjusted as above), to indicate that fewer than the indicated number of moduli are required. */
   if (found==2)
      size=-1*(s+2-omod);
   return 0;
/* This procedure exports the pgf of a term to a text file for use in other programs, 
such as, for example, graphing programs. */

int exdata()
{
    long i, e, s;
    double t;
    FILE *f3;

    /* Open the file to be exported to. */
    f3=fopen(cname, "wt");
    if (f3==NULL)
    {
        printf("Error! Could not open file!");
        return 1;
    }

    /* In the probability file, seek to the position of the values indicating the
    first and last values of the
    coeff of the current term. */
    fseek(f, curr+1?sizeof(int), 0);

    /* Read the first and last values of the coeff of the current term into s and e, respectively. */
    fread(&s, sizeof(long), 1, f);
    fread(&e, sizeof(long), 1, f);

    /* Print these values, in text form, to the export file. */
    fprintf(f3, "\n\%ld", s);
    fprintf(f3, "\n\%ld", e);

    /* Loop through all coefficients of the pgf for the coefficient of the current
    term, and print them to the export file. */
    for (i=s; i<=e; i++)
    {
        fread(&t, sizeof(double), 1, f);
        fprintf(f3, "\n\%8.14f", t);
    }

    /* Close the export file. */
    fclose(f3);
}

/* This procedure is called to display a list of suggested moduli */

int displmod()
{
    char mstr[100], tempstr[50];
    int i, j, nn, temp, l, s;

    /* If no suggestion is possible, display "Not available." */
    if (n==1 || size==-1)
        printf("N/A");

/"Initialize the list of moduli"/
strcpy(mstr," ");

"Set an offset into the mod array."
size=abs(size)+omod-3;

"If fewer than the stated number of moduli are required, calculate how many are required."
if (size<n)
{
    temp=1;
nn=0;
    while (temp<=mod)
    {
        temp=temp*mod[i][nn];
nn=nn+1;
    }
}
else

    nn=n;

"Concatenate the required moduli onto the list of recommended moduli, separated
by commas."
for (i=0;i<nn;i++)
{
    sprintf(tempstr,"%i",mod[i][i]);
    strcat(mstr,tempstr);
    strcat(mstr," ");
}

"Delete the trailing comma from the list."
l=strlen(mstr);
p=mstr-1-2;
"p=0;

printf("%s",mstr);
}

" This procedure is called if the WORST button in the information retrieval window is
pressed."

int infoworstproc()
{
    /* If the WORST button is enabled, set the current position in the probability
    file to the position of the worst case
coefficient, set the current term # to the # of the worst case coefficient, and
display the error and symbolic
    representation for this worst case term.*/

    if (go==1)
    {
        curr=wcurr;
cj=wj;
gextra();
    }
    return 0;
}
/* This procedure is called if the PREVIOUS button in the information retrieval window is pressed. */

int infoprevproc()
{

    /* If the PREVIOUS button is enabled, and there is a previous term, set the current term number to that of the previous term, set the offset of the current term to that of the previous term, and display the error and symbolic representation of the previous term. */

    if (go==1 && cj>0)
    {
        cj=cj-1;
        curr=indx[cj];
        geterr();
    }
    return 0;
}

/* This procedure is called if the NEXT button in the information retrieval window is pressed. Similar to above. */

int infonextproc()
{
    if (go==1 && indx[cj+1]!=-1)
    {
        cj=cj+1;
        curr=indx[cj];
        geterr();
    }
    return 0;
}

/* This procedure is called if the DONE button in the information retrieval window is pressed. */

int infodoneproc()
{
    /* Close the probability file, close the information retrieval window, and reactivate the information button. */

    fclose(f);
    xv_set(infoframe,XV_SHOW,FALSE,NULL);
    xv_set(infobutton,PANEL_INACTIVE,FALSE,NULL);
    return 0;
}

/* This procedure is called if the GO button in the information retrieval window is pressed. */
int infogoproc()
{
  char tstr[40]; /* Temporary string used in converting numbers to strings for output. */
  char *end; /* Pointer used in conversion of strings to numbers. */
  int ckb; /* Value of choice item indicating inputs. */

  ckb=xv_get(tcheckb, PANEL_VALUE); /* Get value of choice item to find out what inputs are specified. */
  ckt=xv_get(tcheckbx, PANEL_VALUE);
  if (ckt==0)
    omod=0;
  else
    if (ckt==1)
      omod=4;
    else
      if (ckt==2)
        omod=8;
      else
        omod=22;

  /* Get value of least significant bit. and display message if it is in error. */
  lsb=xv_get(lsbtext, PANEL_VALUE);
  if (lsb<0 || lsb>mpow)
  {
    printf("Invalid value for least significant bit!");
    return;
  }

  /* If the only input is the product of the moduli. */
  if (ckb ==1)
  {
    /* Get the value of the product of the moduli from the window, and store it in mmod. */
    strcpy(tstr, xv_get(modtext, PANEL_VALUE));
    mmod=strtol(tstr, &end, 10);

    /* Display a message if an invalid value has been entered. */
    if (mmod<=0)
    {
      printf("Invalid product of moduli!");
      go=0;
      return 0;
    }

    /* Find out the maximum error and the term it corresponds to. */
    GetWorst();

    /* Display the worst case term and its error. */
    DisplBit();
/ * Set the fields for number and size of moduli to "Not available."
 */
 xv_set(numtext, PANEL_VALUE, "N/A", NULL);
 xv_set(sizetext, PANEL_VALUE, "N/A", NULL);

 /* Set the list of recommended moduli to "Not available."
 */
 xv_set(mlisttext, PANEL_LABEL_STRING, "N/A", NULL);

 /* Display the confidence percentage in the appropriate field.*/
 sprintf(tstr, "%f", (1-merr)*100);
 xv_set(errtext, PANEL_VALUE, tstr, NULL);

 /* Enable PREVIOUS, NEXT, and WORST buttons.*/
 go=1;

 return 0;

} /* Same as above, but maximum size of moduli also specified.*/
if (ckb==9)
{
    strcpy(tstr, xv_get(modtext, PANEL_VALUE));
    mmod=strtol(tstr, &end, 10);

    if (mmod<=0)
    {
        notice_prompt(infopanel, NULL,
               NOTICE_MESSAGE_STRINGS,
               "Invalid product of moduli!", NULL,
               NOTICE_BUTTON, "OK", 100,
               NULL);
        go=0;
        return 0;
    }

    GetWorst();
    DisplBit();

    /* Get maximum size of moduli from the window, and check for invalid values.*/
    strcpy(tstr, xv_get(sizetext, PANEL_VALUE));
    size=strtol(tstr, &end, 10);

    if (ckt<2 & (size<3 || size>5))
    {
        notice_prompt(infopanel, NULL,
               NOTICE_MESSAGE_STRINGS,
               "# of bits must be 3, 4, 5, or 6!", NULL,
               NOTICE_BUTTON, "OK", 100,
               NULL);
        go=0;
        return 0;
    }

    if (ckt>1 & (size<3 || size>16))
    {

    }
notice_prompt(infopanel, NULL, 
    NOTICE_MESSAGE_STRINGS, 
    "# of bits must be 3, 4, 5, or 6!", NULL,
    NOTICE_BUTTON, "OK", 100,
    NULL);
go=0;
return 0;
}

/* Get required number of moduli.*/
findsize();

/* If the returned value is -1, no such number exists, so display NO,
else display the number of moduli required.*/
if (n==1)
    strcpy(tstr,"NO");
else
    sprintf(tstr,"%i",n);

xv_set(numtext,PANEL_VALUE,tstr,NULL);

/* Generate list of recommended moduli.*/
displmod();

sprintf(tstr,"%.1f",(1-merr)*100);
xv_set(errtext,PANEL_VALUE,tstr,NULL);
go=1;
return 0;
}

/* Same as above, but number of moduli is specified rather than maximum size.*/
if (ckbs==5)
{
    strcpy(tstr,xv_get(modtext,PANEL_VALUE));
    mmod=strtol(tstr,&end,10);

    if (mmod<=0)
    {
        printf("Invalid product of moduli!\n");
go=0;
        return 0;
    }

    GetWorst();
    DisplBit();

    /* Get number of moduli required from window. Store it in n, and print
    error message if value is invalid.*/
    strcpy(tstr,xv_get(numtext,PANEL_VALUE));
n=strtol(tstr,&end,10);

    if (n<1)
{  
    printf("One or more moduli must be employed!");  
    go=0;  
    return 0;  
}

"Get size of moduli required."
findnum();

"Case no such size exists."
if (size==−1)
  strcpy(tstr, "No");
else
  "Case size found, and stated number of moduli is required."
  if(size>0)
    sprintf(tstr, "%i", size);
  else
    "Case fewer than specified number of moduli are required -- append 'to
    value."
    {
      sprintf(tstr, "%i", −l∗size);
      strcat(tstr, " to");
    }

"Display in the size field, the string produced above."
xv_set(size_text, PANEL_VALUE, tstr, NULL);

"Generate list of recommended moduli."
dispmod();

sprintf(tstr, "%f", (l−merr)∗100);
xv_set(err_text, PANEL_VALUE, tstr, NULL);

go=1;
return 0;
}

"In this case, both the size and number of moduli have been input."
if (ckb == 12)
{
    "Get the number of moduli required from the window."
    strcpy(tstr, xv_get(num_text, PANEL_VALUE));
    n=strtol(tstr, &end, 10);

    if (n<1)
    {
      notice_prompt(infopanel, NULL,
                    NOTICE_MESSAGE_STRINGS,
                    "One or more moduli must be employed!", NULL,
                    NOTICE_BUTTON, "OK", 100,
                    NULL);
    }
go=0;
return 0;
}

/* Get the size of moduli required from the window. */
strcpy(tstr, xv_get(sizetext, PANEL_VALUE));
size=strtol(tstr, &end, 10);

if (ckt<2 && (size<3 || size>6))
{
    notice_prompt(infopanel, NULL,
    NOTICE_MESSAGE_STRINGS,
    "# of bits must be 3, 4, 5, or 6!", NULL,
    NOTICE_BUTTON, "OK", 100,
    NULL);
go=0;
return 0;
}

if (ckt>1 && (size<3 || size>16))
{
    notice_prompt(infopanel, NULL,
    NOTICE_MESSAGE_STRINGS,
    "# of bits must be 3, 4, 5, or 6!", NULL,
    NOTICE_BUTTON, "OK", 100,
    NULL);
go=0;
return 0;
}

/* Loop through designated list of moduli to calculate maximum product of
moduli allowed. */
i=0;
mmod=1;
while((i<n && mod[size-3+omod][i]!=-1 && sfact(mod[size-3+omod][i])>mexp)
{
    mmod=mmod*mod[size-3+omod][i];
i++;
}

/* Display this value in the product of moduli field, appending a " if
fewer than n moduli of the given type exit. */
sprintf(tstr, "%i", mmod);

if (i<n)
    strcat(tstr, " ");

xv_set(modtext, PANEL_VALUE, tstr, NULL);

/* Get worst case term and associated error. */
GetWorst();
/* Display the symbolic representation of this term, and its associated error. */
DisplBit();

/* Generate list of recommended moduli. */
displmod();

/* Display the confidence percentage. */
printf(tstr,"%f",(1-merr)*100);
xv_set(errtext,PANEL_VALUE,tstr,NULL);

/* Enable the PREVIOUS, NEXT, and WORST buttons. */
go=1;
return;
}

/* In this case, only the confidence percentage is specified. */

if (ckb==2)
{


/* Get the confidence percentage from the window, and from it, calculate the maximum allowable error. */

strcpy(tstr,xv_get(errtext,PANEL_VALUE));
merr=100-strtold(tstr,&end);
merr=merr/100;

/* Check for invalid values. */

if (merr<0 || merr>100)
{
    printf("Invalid confidence percentage! ")
    go=0;
    return 0;
}

/* Find out minimal product of moduli required to obtain error less than maximum error. */

if (GetWorstP()==1)
{
    go=0;
    return 0;
}

/* Temporarily store the maximum allowable in perr. */
perr=merr;

/* Using the product of moduli calculated above, calculate the worst case term, and its error. */
GetWorst();

/* Restore the maximum allowable error. */
merr=perr;

/* Display the worst case bit and its error. */
geterr();

/* Set the product of moduli fields, and the number and size of moduli fields. */
printf(tstr,"%i",mmod);
xv_set(modtext,PANEL_VALUE,tstr,NULL);
xv_set(numtext,PANEL_VALUE,"N/A",NULL);
xv_set(sizetext,PANEL_VALUE,"N/A",NULL);

/* Set the list of recommended moduli to "Not available." */
xv_set(mlisttext,PANEL_LABEL_STRING,"N/A",NULL);

go=1;
return;
}

/* Same as above, but maximum size of moduli also specified. */
if (chkx=10)
{

   /* Calculate maximum allowable error. */
   strcpy(tstr,xv_get(errtext,PANEL_VALUE));
merr=100-strtol(tstr,&end,10);
merr=merr/100;

   /* Check for invalid value. */
   if (merr<0 || merr>100)
   {
      printf("Invalid confidence percentage!");
go=0;
return 0;
   }

   /* Get required product of moduli to generate confidence */
   if (GetWorstP()==1)
   {
      go=0;
return 0;
   }

   /* Store maximal allowable error, get worst case term and its error, restore maximal allowable error, then display worst case term and its error. */
   perr=merr;
GetWorst();
merr=perr;
geterr();

   /* Get maximum size of moduli from window. */
strcpy(tstr, xv_get(sizetext, PANEL_VALUE));
size=strtol(tstr, &end, 10);

/* Check for invalid value. */

if (ckt<2 || (size<3 || size>6))
{
    printf("# of bits must be 3, 4, 5, or 6!\n");
go=0;
    return 0;
}

if (ckt>1 || (size<3 || size>16))
{
    printf("# of bits must be 3, 4, 5, or 6!\n");
go=0;
    return 0;
}

/* Find out required number of moduli. If -1 is returned, no such number exists, so display NO, else display required number. */
findsize();
if (n=-1)
    strcpy(tstr, "NO");
else
    sprintf(tstr, "%i", n);
xv_set(numtext, PANEL_VALUE, tstr, NULL);

/* Display required product of moduli. */
sprintf(tstr, "%li", nmod);
xv_set(modtext, PANEL_VALUE, tstr, NULL);

/* Generate list of recommended moduli. */
dispmod();

/* Enable PREVIOUS, NEXT, and WORST buttons. */
go=1;
    return 0;
}

/* Same as above, but number of moduli is specified rather than size. */

if (ckb==6)
{
    /* Calculate the maximal allowable error from the confidence % specified. */
    strcpy(tstr, xv_get(errtext, PANEL_VALUE));
    merr=100-strtod(tstr, &end);
    merr=merr/100;

    /* Check for invalid values. */
if (merr<0 || merr>100)
{
    notice_prompt(infopanel, NULL,
        NOTICE_MESSAGE_STRINGS,
        "Invalid confidence percentage!", NULL,
        NOTICE_BUTTON, "OK", 100,
        NULL);
    go=0;
    return 0;
}

/* Find out required product of moduli. */
if (!GetWorst())
{
    go=0;
    return 0;
}

/* Store maximal allowable error, get worst case term and its error, restore maximal allowable error, then display worst case term and its error. */

perr=merr;
GetWorst();
merr=perr;
geterr();

/* Get number of moduli from window. */
strcpy(tstr, xv_get(numtext, PANEL_VALUE));
n=strtoi(tstr, &end, 10);

/* Check for invalid value. */
if (n<1)
{
    printf("One or more moduli must be employed!");
    go=0;
    return 0;
}

/* Find out the required size of moduli. */
findnum();

/* If -1 is returned, no such size exists. else if a negative integer is returned, append a ' to the size, indicating that fewer than the specified number of moduli are required. */
if (size==-1)
    strcpy(tstr,"NO");
else
    if (size>0)
        sprintf(tstr,"%i",size);
    else
    {
        sprintf(tstr,"%i",size-1);
        strcat(tstr," ");
    }
/ * Display the size of moduli required, or the message produced above if no such size was found. */
    xv_set(sizetext, PANEL_VALUE, tstr, NULL);

    /* Display the required product of moduli. */
    sprintf(tstr, "%i", mmmod);
    xv_set(modtext, PANEL_VALUE, tstr, NULL);

    /* Generate list of recommended moduli. */
    displmod();

    /* Enable the PREVIOUS, NEXT, and WORST buttons. */
    go=1;

    return 0;
 }

return 0;

}

float objfunc(reqmod, repfact)
int reqmod;
int repfact:
{
    float temp;

    temp=pow((float)repfact, repwgt);
    temp=temp*(float)reqmod;
    return temp;
}

/* Mainline program */

main(argc, argv)
int argc;
char *argv[];
{
    remove ("temp.pgf");
    remove("distr0.tmp");
    remove("distr1.tmp");
    remove("distr2.tmp");
    remove("distr3.tmp");
    exit(0);
Appendix C

Verilog Code

C.1 Library Elements

C.1.1 Adders

1-bit adder

// Verilog HDL for "HDLib", "adder1" "behavioral"

module adder1 (CXOUT, S, CXIN, a, b, CK);
        output CXOUT;
        output S;
        input CXIN;
        input a;
        input b;
        input CK;

        wire CIN, COUT;

        assign #2 { COUT, S } = (CK==1)? a+b+CIN:0;
        not ciinv(CIN, CXIN);
        coinv(CXOUT,COUT);

endmodule

4-bit adder with carry

// Verilog HDL for "DFMT", "adder4c" "behavioral"

`timescale 1ns/10ps
module adder4c (CXOUT, S, CIN, a, b,CK);
        output CXOUT;
        output [3:0] S;
        input CIN;
        input [3:0] a;
        input [3:0] b;

endmodule
input CK;

wire COUT;

assign #1 { COUT, S } = (CK==1)?a+b+CIN:0;
not coinv(CXOUT,COUT);

endmodule

8-bit adder

// Verilog HDL for "DFMT", "adder8" "_behavioral"

module adder8 (CXOUT, S, CXIN, a, b, CK);
ooutput CXOUT;
ooutput [7:0] S;
ininput CXIN;
ininput [7:0] a;
ininput [7:0] b;
ininput CK;

wire CIN, COUT;

assign #2 { COUT, S } = (CK==1)?a+b+CIN:0;
not coinv(CXOUT,COUT),
ciinv(CIN,CXIN);

endmodule

8-bit adder with carry

// Verilog HDL for "DFMT", "adder8c" "_behavioral"

module adder8c (CXOUT, S, CXIN, a, b, CK);
ooutput CXOUT;
ooutput [7:0] S;
ininput CXIN;
ininput [7:0] a;
ininput [7:0] b;
ininput CK;

wire CIN, COUT;

assign #2 { COUT, S } = (CK==1)?a+b+CIN:0;
not coinv(CXOUT,COUT),
ciinv(CIN,CXIN);

endmodule

8 bit adder no carry
module adder8nc (S, a, b, CK) ;
output [7:0] S;
input a;
input [7:0] b;
input CK;
assign #2 S = (CK==1) ? a+b:0;
endmodule

module adderm (COUT, S, CIN, a, b, CK) ;
	//timescale ns/10ps
	//module adder4c (COUT, S, CIN, a, b, CK):
	output COUT;
	output [3:0] S;
	input CIN;
	input [3:0] a;
	input [3:0] b;
	input CK;

	wire COUT;

assign #2 { COUT, S } = (CK==1) ? a+b+CIN:0;
//not coinv(CXOUT,COUT);
endmodule

**Pipelined adder block**

module ADD_BLOCK ( A3OUT, ACC_OUT, AOOUT, B3, A3IN, ACC_IN, AIN, B_in, CK, CK_coef );
output B3;
input B_in, CK, CK_coef;
output [9:0] AOOUT;
output [1:0] A3OUT;
output [15:0] ACC_OUT;
input [15:0] ACC_IN;
input [1:0] A3IN;
input [9:0] AIN;

// Buses in the design
wire [0:1] net360;
wire [0:1] net490;
wire [0:15] net39;
wire [0:15] net199;
wire [0:1] net400;
wire [0:15] net119;
wire [0:9] net268;
wire [0:1] net420;
wire [0:9] net58;
wire [0:9] net28;
wire [0:9] net148;
wire [0:9] net198;
wire [0:1] net250;
wire [0:1] net350;
wire [0:9] net388;
wire [0:1] net180;
wire [0:1] net10;
wire [0:15] net259;
wire [0:9] net108;
wire [0:1] net530;
wire [0:1] net130;
wire [0:9] net78;
wire [0:1] net240;
wire [0:15] net359;
wire [0:1] net460;
wire [0:15] net189;
wire [0:1] net190;
wire [0:15] net219;
wire [0:9] net508;
wire [0:9] net488;
wire [0:1] net500;
wire [0:9] net408;
wire [0:9] net398;
wire [0:15] net59;
wire [0:15] net109;
wire [0:15] net459;
wire [0:1] net120;
wire [0:9] net48;
wire [0:9] net128;
wire [0:1] net50;
wire [0:15] net319;
wire [0:9] net538;
wire [0:9] net478;
wire [0:15] net99;
wire [0:1] net210;
wire [0:15] net209;
wire [0:9] net338;
wire [0:9] net368;
wire [0:15] net309;
wire [0:1] net430;
wire [0:9] net218;
wire [0:9] net248;
wire [0:1] net280;
wire [0:15] net289;
wire [0:1] net290;
wire [0:15] net429;
wire [0:9] net518;
wire [0:1] net510;
wire [0:1] net220;
wire [0:9] net128;
wire [0:1] net440;
wire [0:9] net428;
wire [0:15] net269;
wire [0:1] net410;
wire [0:9] net38;
wire [0:15] net529;
wire [0:1] net110;
wire [0:1] net90;
wire [0:15] net329;
wire [0:9] net178;
wire [0:1] net450;
wire [0:15] net149;
wire [0:1] net230;
wire [0:15] net69;
wire [0:1] net370;
wire [0:15] net409;
wire [0:9] net138;
wire [0:15] net29;
wire [0:9] net308;
wire [0:1] net340;
wire [0:1] net510;
wire [0:15] net489;
wire [0:1] net140;
wire [0:1] net390;
wire [0:15] net469;
wire [0:9] net98;
wire [0:15] net249;
wire [0:9] net358;
wire [0:1] net200;
wire [0:1] net110;
wire [0:15] net179;
wire [0:15] net379;
wire [0:1] net70;
wire [0:15] net499;
wire [0:15] net349;
wire [0:1] net60;
wire [0:1] net540;

specify
    specparam CDS_LIBNAME = "HDLib";
    specparam CDS_CELLNAME = "ADD_BLOCK";
    specparam CDS_VIEWNAME = "schematic";
endspecify

PIPE_ADD I54 ( A[15:0], ACC_OUT[15:0], A[9:0], B3, net30[0:1],
               net28[0:9], net29[0:15], net27, CK, CK_coef);
PIPE_ADD I53 ( net30[0:1], net29[0:15], net28[0:9], net27, net40[0:1],
               net38[0:9], net39[0:15], net37, CK, CK_coef);
PIPE_ADD I52 ( net40[0:1], net39[0:15], net38[0:9], net37, net50[0:1],
               net48[0:9], net49[0:15], net47, CK, CK_coef);
Check block

// Library - HDLib, Cell - check, View - schematic
// LAST TIME SAVED: Jun 29 11:54:22 1999
// NETLIST TIME: Jun 29 17:20:35 1999
.timescale lns / 10ps

module check (out, in);
output out;
input [11:0] in;
endmodule
specify
  specparam CDS_LIBNAME = "HDLib";
  specparam CDS_CELLNAME = "check";
  specparam CDS_VIENNAME = "schematic";
endspecify

inv_in I11 ( net3, in[10]);
inv_in I7 ( net5, in[6]);
inv_in I6 ( net7, in[7]);
inv_in I3 ( net9, in[2]);
inv_in I2 ( net11, in[1]);
inv_in I1 ( net13, in[0]);
wand2_1 I15 ( out, net30, net23);
wand2_1 I15 ( net20, net29, net26);
wand2_1 I14 ( net23, net35, net32);
wand2_1 I13 ( net26, net41, net38);
wand2_1 I12 ( net29, net44, net47);
wand2_1 I10 ( net32, net3, in[11]);
wand2_1 I9 ( net35, in[8], in[9]);
wand2_1 I8 ( net38, net5, net7);
wand2_1 I5 ( net41, in[4], in[5]);
wand2_1 I4 ( net44, net13, net11);
wand2_1 I0 ( net47, net9, in[3]);
endmodule

C.1.2 Latches

1-bit latch

// Verilog HDL for "FABR", "dff_in" "_behavioral"

module dff_in ( O, CK, I);
  output O;
  input CK;
  input I;

  reg O;

  always @(nedge CK)
    O=I;
endmodule

2 cascaded 1-bit latch

// Library - HDLib, Cell - dff_inX2, View - schematic
`timescale 1ns / 10ps

module dff_inX2 ( O, CK, I );

output O;
input CK, I;
specify
  specparam CDS_LIBNAME = "HDLlib";
  specparam CDS_CELLNAME = "diff_inX2";
  specparam CDS_VIETNAME = "schematic";
endspecify

dff_in I1 ( 0, CK, net7);
dff_in I0 ( net7, CK, I);

endmodule

static latch (1 bit)

// Verilog HDL for "HDLib", "diff_in" "behavioral"

module static_dff (O, CK, I);
  output O;
  input CK;
  input I;

  reg O;

  always @(CK==1)
    #5 O=I;

endmodule

2-bit latch

// Verilog HDL for "HDLib", "diff2_in" "behavioral"

module dff2_in (O, CK, I);
  output [1:0] O;
  input CK;
  input [1:0] I;

  reg [1:0] O;

  always @(negedge CK)
    O=I;

endmodule

2 cascaded 2-bit latch

// Library - HDLib, Cell - dff2_inX2, View - schematic
// LAST TIME SAVED: Jun 30 11:24:09 1999
// NETLIST TIME: Jun 30 11:25:51 1999
'timescale 1ns / 10ps
module dff2_inX2 (Out, CK, In);

input CK;
output [1:0] Out;
input [1:0] In;

// Buses in the design
wire [0:1] net12;

specify
  specparam CDS_LIBNAME = "HDLib";
specparam CDS_CELLNAME = "dff2_inX2";
specparam CDS_VIEWNAME = "schematic";
endspecify
dff2_in I1 (Out[1:0], CK, net12[0:1]);
dff2_in I0 (net12[0:1], CK, In[1:0]);
endmodule

3 cascaded 2-bit latch

// Library - HDLib, Cell - dff2_inX3, View - schematic
// LAST TIME SAVED: May 19 14:20:54 1999
// NETLIST TIME: May 19 14:27:09 1999
timescale 1ns, 10ps

module dff2_inX3 (Out, CK, In);

input CK;
output [1:0] Out;
input [1:0] In;

// Buses in the design
wire [0:1] net9;
wire [0:1] net12;

specify
  specparam CDS_LIBNAME = "HDLib";
specparam CDS_CELLNAME = "dff2_inX3";
specparam CDS_VIEWNAME = "schematic";
endspecify
dff2_in I2 (Out[1:0], CK, net9[0:1]);
dff2_in I1 (net9[0:1], CK, net12[0:1]);
dff2_in I0 (net12[0:1], CK, In[1:0]);
endmodule
4-bit latch

// Verilog HDL for "HDLib", "dff4_in" "behavioral"

module dff4_in (O, CK, I);
output [3:0] O;
input CK;
input [3:0] I;
reg [3:0] O;

always @ (negedge CK)
O=I;
endmodule

8-bit latch

// Verilog HDL for "HDLib", "dff8_in" "behavioral"

module dff8_in (O,CK,I);
output [7:0] O;
input [7:0] I;
input CK;
reg [7:0] O;

always @ (negedge CK)
O=I;
endmodule

2 cascaded 8-bit latch

// Library - HDLib, Cell - dff8_inX2, View - schematic
// LAST TIME SAVED: Mar 1 14:47:11 1999
// NETLIST TIME: Mar 1 15:02:41 1999
`timescale ns / 10ps

module dff8_inX2 ( O, CK, I );

input CK;
output [7:0] O;
input [7:0] I;

// Buses in the design
wire [0:7] net6;

specify
    specparam CDS_LIBNAME = "HDLib";
specparam CDS_CELLNAME = "dff8_inX2";
specparam CDS_VIEWNAME = "schematic";
endspecify

dff8_in I1 ( O[7:0], CK, net6[0:7]);
dff8_in I0 ( net6[0:7], CK, I[7:0]);
endmodule

3 cascaded 8-bit latch

// Library - HLib, Cell - dff8_inX3, View - schematic
// LAST TIME SAVED: Mar 1 14:58:23 1999
// NETLIST TIME: Mar 1 15:02:41 1999
	timescale 1ns / 10ps
module dff8_inX3 ( O, CK, I );
input CK;
output [7:0] O;
input [7:0] I;

// Buses in the design
wire [0:7] net9;
wire [0:7] net12;

specify
	specparam CDS_LIBNAME = "HLib";
specparam CDS_CELLNAME = "dff8_inX3";
specparam CDS_VIEWNAME = "schematic";
endspecify

dff8_in I2 ( O[7:0], CK, net9[0:7]);
dff8_in I1 ( net9[0:7], CK, net12[0:7]);
dff8_in I0 ( net12[0:7], CK, I[7:0]);
endmodule

9-bit latch

// Verilog HDL for "HLib", "dff9_in" "behavioral"

module dff9_in (O,CK,I);
output [8:0] O;
input [8:0] I;
input CK;
reg [8:0] O;
always @(negedge CK)
O = I;
O=I;
endmodule

2 cascaded 9-bit latch

// Library - HDLib, Cell - dff9_inX2, View - schematic
timescale ins / 10ps

module dff9_inX2 ( O, CK, I );

input CK;
output [8:0] O;
input [8:0] I;

// Buses in the design
wire [0:8] net39;

specify
specparam CDS_LIBNAME = "HDLib";
specparam CDS_CELLNAME = "dff9_inX2";
specparam CDS_VIEWNAME = "schematic";
endspecify
dff9_in I1 ( O[8:0], CK, net39[0:8]);
dff9_in I0 ( net39[0:8], CK, I[8:0]);
endmodule

3 cascaded 9-bit register

// Library - HDLib, Cell - dff9_inX3, View - schematic
timescale ins / 10ps

module dff9_inX3 ( O, CK, I );

input CK;
output [8:0] O;
input [8:0] I;

// Buses in the design
wire [0:8] net39;
wirer [0:8] net35;

specify
specparam CDS_LIBNAME = "HDLib";
specparam CDS_CELLNAME = "dff9_inX3";
specparam CDS_VIEWNAME = "schematic";
endspecify
dff9_in I2 ( O[8:0], CK, net35[0:8]);
dff9_in I1 ( net35[0:8], CK, net38[0:8]);
dff9_in I0 ( net38[0:8], CK, I[8:0]);
endmodule

10-bit latch

// Verilog HDL for "HDLib", "dff10_in" "behavioral"

module dff10_in (C,CK,I),
    output [9:0] O;
inpu[9:0] I;
inpu CK;
reg [9:0] O;
always @ (negedge CK)
    O=I;
endmodule

two cascaded 10-bit latches

// Library - HDLib, Cell - dff10_inX2, View - schematic
// LAST TIME SAVED: Jun 30 11:22:00 1999
// NETLIST TIME: Jun 30 11:25:51 1999
`timescale 1ns / 10ps

module dff10_inX2 ( Out, CK, In );

input CK;
output [9:0] Out;
input [9:0] In;

// Buses in the design
wire [0:9] net12;

specify
    specparam CDS_LIBNAME = "HDLib";
specparam CDS_CELLNAME = "dff10_inX2";
specparam CDS_VIEWNAME = "schematic";
endspecify
dff10_in I1 ( Out[9:0], CK, net12[0:9]);
dff10_in I0 ( net12[0:9], CK, In[9:0]);
endmodule

3 cascaded 10-bit latch
module dff10_inX3 ( Out, CK, In );
  input CK;
  output [9:0] Out;
  input [9:0] In;
  // Buses in the design
  wire [0:9] net9;
  wire [0:9] net12;

specify
  specparam CDS_LIBNAME = "HDLib";
  specparam CDS_CELLNAME = "dff10_inX3";
  specparam CDS_VIEWNAME = "schematic";
endspecify

dff10_in I2 ( Out[9:0], CK, net9[0:9]);
dff10_in I1 ( net9[0:9], CK, net12[0:9]);
dff10_in I0 ( net12[0:9], CK, In[9:0]);
endmodule

16-bit latch

// Verilog HDL for "HDLib", "dff16_in" "behavioral"

module dff16_in (O,CK,I);
  output [15:0] O;
  input [15:0] I;
  input CK;
  reg [15:0] O;
  always @ (negedge CK)
    O = I;
endmodule

3 cascaded 16-bit latch

// Library - HDLib, Cell - dff16_inX3, View - schematic
// LAST TIME SAVED: May 19 11:44:37 1999
// NETLIST TIME: Jun 29 12:03:58 1999
\timescale 1ns / 10ps
module dff16_inX3 (Out, CK, In);

input CK;
output [15:0] Out;
input [15:0] In;

//-- Buses in the design

wire [0:15] net9;
wire [0:15] net12;

specparam CDS_LIBNAME = "HDLib";
specparam CDS_CELLNAME = "dff16_inX3";
specparam CDS_VIENNAME = "schematic";
endspecify

dff16_in I2 (Out[15:0], CK, net9[0:15]);
dff16_in I1 (net9[0:15], CK, net12[0:15]);
dff16_in I0 (net12[0:15], CK, In[15:0]);
endmodule

10-bit shift register (serial in serial out)

//-- Library - HDLib, Cell - dff10_chain, View - schematic
//-- LAST TIME SAVED: Jun 29 17:19:50 1999
//-- NETLIST TIME: Jun 29 17:20:35 1999
	timescale ns / 10ps

module dff10_chain (bl, b3, out, CK, in);
output bl, b3;
input CK, in;
inout [9:0] out;

specify
specparam CDS_LIBNAME = "HDLib";
specparam CDS_CELLNAME = "dff10_chain";
specparam CDS_VIENNAME = "schematic";
endspecify

dff_in I10 (out[9], CK, out[8]);
dff_in I11 (bl, CK, out[9]);
dff_in I9 (out[8], CK, out[7]);
dff_in I0 (out[0], CK, in);
dff_in I1 (out[1], CK, out[0]);
dff_in I2 (out[2], CK, out[1]);
dff_in I3 (out[3], CK, out[2]);
dff_in I4 (out[4], CK, out[3]);
dff_in I5 (out[5], CK, out[4]);
dff_in I6 (out[6], CK, out[5]);
dff_in I7 (out[7], CK, out[6]);
dff_in I8 (b3, CK, b1);
endmodule
serial in, parallel out 10-bit shift register

// Library - HDLib, Cell - dff10_chain, View - schematic
// NETLIST TIME: May 18 15:33:12 1999
' timescale 1ns / 10ps

module dff10_chain ( b1, b3, (out[8], out[7], out[6], out[5], out[4],
  out[3], out[2], out[1], out[0]), CK, in );
output b1, b3;
input CK, in;
inout [0:9] out;

specify
  specparam CDS_LIBNAME = "HDLib";
  specparam CDS_CELLNAME = "dff10_chain";
  specparam CDS_VIENAME = "schematic";
endspecify

static_dff II0 ( out[9], CK, out[8]);
static_dff II1 ( b3, CK, out[9]);
static_dff II9 ( out[8], CK, out[7]);
static_dff I00 ( out[0], CK, in);
static_dff I11 ( out[1], CK, out[0]);
static_dff I22 ( out[2], CK, out[1]);
static_dff I33 ( out[3], CK, out[2]);
static_dff I44 ( out[4], CK, out[3]);
static_dff I55 ( out[5], CK, out[4]);
static_dff I66 ( out[6], CK, out[5]);
static_dff I77 ( out[7], CK, out[6]);
static_dff I88 ( b1, CK, b3);
endmodule

9-bit shift register

// Library - HDLib, Cell - dff9_chain, View - schematic
// LAST TIME SAVED: May 14 11:10:02 1999
// NETLIST TIME: Jun 29 12:04:04 1999
' timescale 1ns / 10ps

module dff9_chain ( output_, out, CK, in );
output output_;
input CK, in;
inout [0:7] out;

specify
  specparam CDS_LIBNAME = "HDLib";
  specparam CDS_CELLNAME = "dff9_chain";
  specparam CDS_VIENAME = "schematic";
endspecify

dff_in I8 ( output_, CK, out[7]);
dff_in I7 ( out[7], CK, out[6]);
C.1.3 Multiplexers

10-bit multiplexer

```verbatim
// Verilog HDL for "HDLib", "mux10" "behavioral"

module mux10 (Q, DNZ, DZ, NZ, NZ, CK);
  output [9:0] Q;
  input [9:0] DNZ;
  input [9:0] DZ;
  input NZ;
  input Z;
  input CK;

  wire [9:0] Q_INT;

  assign #1 Q_INT=(NZ==1)?DNZ:DZ;
  assign Q=(CK==1)?Q_INT:0;
endmodule
```

10-to-3 multiplexer

```verbatim
// Verilog HDL for "HDLib", "mux10_3" "behavioral"

module mux10_3(Q, D1, D2, D3, Z1, Z2, Z3, CK);
  output [9:0] Q;
  input [9:0] D1;
  input [9:0] D2;
  input [9:0] D3;
  input Z3;
  input Z1;
  input Z2;
  input CK;

  wire [9:0] Q_INT;

  assign #1 Q_INT=(Z1==1)?D1:((Z2==1)?D2:((Z3==1)?D3:0));
  assign Q=(CK==1)?Q_INT:0;
endmodule
```
2 to 3 multiplexer

// Verilog HDL for "HDLib", "mux2_3" "behavioral"

module mux2_3 (Q, D1, D2, D3, Z1, Z2, Z3, CK);
    output [1:0] Q;
    input [1:0] D1;
    input [1:0] D2;
    input [1:0] D3;
    input Z1;
    input Z2;
    input Z3;
    input CK;

    wire [1:0] Q_INT;

    assign #1 Q_INT=(Z1==1)?D1:((Z2==1)?D2:((Z3==1)?D3:0));
    assign Q=(CK==1)?Q_INT:0;
endmodule

9-bit multiplexer

// Verilog HDL for "DFMT", "mux9" "behavioral"

module mux9 (Q, DNZ, DZ, NZ, Z, CK);
    output [8:0] Q;
    input [8:0] DNZ;
    input [8:0] DZ;
    input NZ;
    input Z;
    input CK;

    wire [8:0] Q_INT;

    assign #1 Q_INT=(NZ==1)?DNZ:DZ;
    assign Q=(CK==1)?Q_INT:0;
endmodule

9-bit multiplexer

// Verilog HDL for "DFMT", "mux9_4" "behavioral"

module mux9_4 (Q, D1, D2, D3, D4, Z1, Z2, Z3, Z4, CK);
    output [8:0] Q;
    input [8:0] D1;
    input [8:0] D2;
    input [8:0] D3;
    input [8:0] D4;
    input Z1;
    input Z2;
input z3;
input z4;
input CK;

wire [8:0] Q_INT;

assign #1 Q_INT=(z1==1)?D1:((z2==1)?D2:((z3==1)?D3:((z4==1)?D4:0)));
// assign #1 Q_INT=(z2==1)?D2:0;
// assign #1 Q_INT=(z3==1)?D3:0;
// assign #1 Q_INT=(z4==1)?D4:0;
assign Q=(CK==1) ? Q_INT:0;
// may need case
endmodule

C.1.4 Miscellaneous

8-bit Inverter

// Library - HDLib, Cell - inv8, View - schematic
// LAST TIME SAVED: May 18 14:59:12 1999
// NETLIST TIME: May 18 15:33:12 1999
	timescale 1ns , 10ps

module inv8 ( out, in );

output [7:0] out;
input [7:0] in;

specify
	specparam CDS_LIBNAME = "HDLib"
	specparam CDS_CELLNAME = "inv8"
	specparam CDS_VIEWNAME = "schematic"
endspecify

inv_in I7 ( out[7], in[7] );
inv_in I6 ( out[6], in[6] );
inv_in I5 ( out[5], in[5] );
inv_in I4 ( out[4], in[4] );
inv_in I3 ( out[3], in[3] );
inv_in I2 ( out[2], in[2] );
inv_in I1 ( out[1], in[1] );
inv_in IO ( out[0], in[0] );

endmodule

1-bit inverter

// Verilog HDL for "FABR", "inv_in" "_behavioral"

module inv_in (O, I);
	output O;

input I;

assign #0.5 O=-I;

endmodule

Merge cell

// Verilog HDL for "HDLlib", "merge" "behavioral"

module merge4 (O, I0, I1, I2, I3);
output [3:0] O;
input I2, I3, I1, I0;
trireg I0, I1, I2, I3; //optional declaration
trireg [3:0] data;
reg [3:0] O;
assign
    data=(I3, I2, I1, I0);

initial
begin
    assign O=data;
end
endmodule

Merge cell

// Verilog HDL for "HDLlib", "merge" "behavioral"

module merge8 (O, I0, I1, I2, I3, I4, I5, I6, I7);
output [7:0] O;
input I1, I2, I3, I4, I5, I6, I7;
trireg I1, I2, I3, I4, I5, I6, I7;
trireg [7:0] data;
reg [7:0] O;
assign
    data=(I7, I6, I5, I4, I3, I2, I1, I0);
initial
begin
    assign O=data;
end
endmodule

Bus splitter

// Verilog HDL for "HDLlib", "split" "behavioral"
module split10_3 (O0, O1, O2, I) ;

output [3:0] O0, O1, O2;
input [9:0] I;
trireg [9:0] I;
trireg [3:0] data1, data2, data3;
reg [3:0] O0, O1, O2;
assign
data1=[I[9], I[2], I[1], I[0]],
data2=[I[9], I[5], I[4], I[3]],
data3=[I[9], I[8], I[7], I[6]];

initial
begin
assign O0=data1;
assign O1=data2;
assign O2=data3;
end
endmodule

8-bit two's complement block

:// Verilog HDL for "HDLib", "twos_comp" ~behavioral"

module twos_comp (O, I) ;

output [7:0] O;
input [7:0] I;
assign O=((-I)+1);
endmodule

10-bit 2's complement

:// Library - HDLib, Cell - twos_compl0, View - schematic
:// LAST TIME SAVED: May 20 14:44:49 1999
:// NETLIST TIME: Jun 29 12:03:58 1999
'timescale 1ns / 10ps

module twos_compl0 ( out, CK, in );

input CK;
output [9:0] out;
input [9:0] in;

// Buses in the design
wire [0:7] net20;
specify
    specparam CDS_LIBNAME = "HDLib";
    specparam CDS_CELLNAME = "twos_comp10";
    specparam CDS_VIEWNAME = "schematic";
endspecify

adder1 i24 ( net22, out[0], cds_globals.GND_, cds_globals.GND_, net4, CK);
adder1 i25 ( net15, out[9], net12, cds_globals.GND_, net10, CK);
inv_in i28 ( net4, in[0]);
inv_in i29 ( net10, in[9]);
adder8 i26 ( net12, out[3:1], net22, cds_globals.GND_,
           cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
           cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
           cds_globals.GND_, net20[0:7], CK);
inv8 i27 ( net20[0:7], in[8:1]);
endmodule

**NAN block**

// Verilog HDL for "HDLib", "Nan" behavioral

module NAN (O, I);
    output O;
    input [7:0] I;
    reg temp;
    wire 0;

    assign O = temp;

initial
begin
    if (I==0) temp=1;
    else temp=0;
end
endmodule

**logic and**

// Verilog HDL for "HDLib", "wand2_1" behavioral

module wand2_1 (op, ipl, ip2);
    output op;
    input ipl;
    input ip2;

and gand(op, ipl, ip2);
endmodule
logic nor

// Verilog HDL for "HDLib", "nor2_1" "behavioral"

module nor2_1 (op, ip1, ip2);
    output op;
    input ip1;
    input ip2;

    nor gnor(op, ip1, ip2);
endmodule

logic or

// Verilog HDL for "HDLib", "or2_1" "behavioral"

module or2_1 (op, ip1, ip2);
    output op;
    input ip1;
    input ip2;

    or gor(op, ip1, ip2);
endmodule

C.1.5 Fermat ALUs

Dfmt257

// Verilog HDL for "DFMT", "dfmt257" "behavioral"

`timescale ns/10ps
module dfmt2(CK, IN_COEFAin,Aout,Cin,Cout,CII, CO1,NAin,NAcoef,M1Aout,M1Cin,
M1Cout);
    output [7:0] Aout,Cout;
    input [7:0] IN_COEFAin,Ain,Cin;
    output CO1,NAcout, NAout;
    input CII,NAin,NAcoef, N1Cin;
    input CK;
    reg N1Cout;

    reg [7:0] Aout,AB,Cout;
    reg CO1,NAout, N1ATmp;
    reg [7:0] alout_R, romout_R;

    wire [7:0] alout_T,romout_T,a2out, a3out;
    wire [8:0] MUXout;
    wire a2CX,NZ,NT,Z, C1IN;

    always @ (negedge CK)
    fork


Aout=AB;
AB=Ain;
(CO1,Cout)=MUXout;
NICout=(MUXout==9'b1_0000_0000)?1:0;
NLAout=NLatmp;
NLatmp=NLAin;

// Internal pipeline register
alout_R=alout_T;
romout_R=romout_T;

join
adder8  al(.,alout_T,1'b1,Ain,IN_COEF,CK);
JKCK35  rom(romout_T,alout_R,CK);
adder8c a2(a2CX,a2out,C11,Cin,romout_R,CK);
// adder8c a3(a3CX,alout,C11,Cin,8'b1000_0000,CK);
mux9_4  gmux(MUX-
out,(~a2CX,a2out),(1'b0,romout_R),(C11,Cin),9'b1_0000_0000,21.22.23.24,CK);
nor   gnor1(NZ,NLAin,NIcoef);
not   gnot2(C11N,C11);
and  gand1(21,NZ,Z);
not   gnot2(23,NZ,NIcin);
and  gand2(22,NZ,NIcin);
// nor   gnor4(23,NT,NIcin);
// nor   gnor5(24,NT,NIcin);
not   gnot(Z,NIcin);

endmodule

original Dfmt

// Verilog HDL for "DFM", "dfmct257" "_behavioral"

timescale 1ns/10ps
module dfmct257(CX, IN_COEF,A,B,C,D,C11, C01,NLA,NIB,NIC);
output [7:0] B,D;
input [7:0] IN_COEF,A,C;
output CO1,NIC;
input C11,NLA,NIB;
input CK;

reg [7:0] B,AB,D;
reg CO1,NIC, NLatmp;
reg [7:0] alout_R, romout_R;

wire [7:0] alout_T,romout_T,a2out;
wire [8:0] MUXout;
wire a2CX,NZ;

always @(negedge CK)
fork
B=AB;
AB=A;
(CO1,D)=MUXout;
NIC=NLatmp;
NLatmp=NLA;

endfork
// Internal pipeline register
    alout_R=alout_T;
    romout_R=romout_T;
join

    adder8     al!,alout_T,1'b1,A,IN_COEF,CK);
dRCM256     rom(romout_T,alout_R,CK);
adder8c     a2(a2CX,alout,CIL,C,romout_R,CK);
mux9       gmux(MUXout,(a2CX,a2out),(CIL,C),NZ,Z,CK);
nor         gnor(NZ,NLA,N1B);
not         gnot(Z,NZ);
endmodule

dfmt-block (with serial loading)

// Library - HDLib, Cell - dfmt_block, View - schematic
// LAST TIME SAVED: May 14 11:28:44 1999
// NETLIST TIME: Jun 29 12:04:04 1999
	`timescale lns / 10ps

module dfmt_block ( Aout, C1l, Cout, N1Aout, N1Cout, coef_out, Ain,
      C1l, CK, CK_coef, Cin, NL Ain, N1Cin, coef_in );
output C1l, N1Aout, N1Cout, coef_out;
input C1l, CK, CK_coef, N1Ain, N1Cin, coef_in;
output [7:0] Aout;
output [7:0] Cout;
input [7:0] Ain;
input [7:0] Cin;

// Buses in the design
wire [0:7] net3;

specify
    specparam CDS_LIBNAME = "HDLib";
specparam CDS_CELLNAME = "dfmt_block";
specparam CDS_VIEWNAME = "schematic";
endspecify
dff9_chain I1 ( coef_out, net3[0:7], CK_coef, coef_in);
dfmt2 I0 ( CK, net3[0:7], Ain[7:0], Aout[7:0], Cin[7:0], Cout[7:0],
      C1l, C1l, NL Ain, coef_out, N1Aout, N1Cin, N1Cout);
endmodule

diminished-1 block

// Verilog HDL for "HDLib", "diminished_1" "behavioral"
module diminished_1 ( O, I );
output [8:0] O;
input [7:0] I;
trireg [8:0] temp;
trireg [7:0] temp2;
trireg temp3;
reg [8:0] O;

assign temp=(I[0], I) + 9'b01111111;
    temp2=temp[7:0];
    temp3=~temp[8];
initial
begin
    assign O=(I[7]==1) ? (I[6:0], I).temp2-temp3;
end
endmodule

index mapper block

// Verilog HDL for "HDLib", "index_map" "behavioral"

module dim_index_map (O, CK, I);
output [7:0] O;
input CK;
input [7:0] I;
reg [7:0] rom256[0:255];
initial
    $readmemb("/home/vlsi/mshahka/CMOS35/VER/fwrdrombin.txt", rom256, 0, 255);
assign #2 O=(CK==1) ? rom256[I]:0;
endmodule

ROM lookup table

// Verilog HDL for "DROM", "dROM256" "behavioral"

module dROM256 (D, A, CK);
    output [7:0] D;
    input [7:0] A;
    input CK;
    reg [7:0] rom256[0:255];
    initial
        $readmemb("/home/vlsi/mshahka/CMOS35/VER/revrombin.txt", rom256, 0, 255);
        assign #2 D=(CK==1) ? rom256[A]:0;
endmodule
C.2 Filter description

// Library - HDLib, Cell - FILTER, View - schematic
// LAST TIME SAVED: Jun 29 17:20:01 1999
// NETLIST TIME: Jun 29 17:20:43 1999
`timescale 1ns / 10ps

module FILTER ( ACC_OUT, OUT0, OUT1, OUT2, OUT3, OUT4, out_coef, CK,
               CK_coef, In, coef_in );

output out_coef;
input CK, CK_coef, coef_in;
output [15:0] ACC_OUT;
output [7:0] OUT1;
output [7:0] OUT4;
output [7:0] OUT0;
output [7:0] OUT1;
output [7:0] OUT2;
input [9:0] In;

// Buses in the design

wire [11:0] x;
wire [1:0] Ai3IN;
wire [0:3] net56;
wire [0:9] net72;
wire [0:9] net67;
wire [0:8] net8;
wire [0:8] net52;
wire [0:8] net7;
wire [0:8] net10;
wire [0:3] net55;
wire [0:8] net9;
wire [0:8] net61;
wire [0:1] net68;
wire [0:8] net6;
wire [0:3] net54;
wire [0:8] net60;
wire [0:9] net138;
wire [0:9] net66;
wire [0:8] net64;
wire [0:8] net61;

specify
    specparam CDS_LIBNAME = "HDLib";
    specparam CDS_CELLNAME = "FILTER";
    specparam CDS_VIEWNAME = "schematic";
endspecify

check II5 ( out_coef, x[11:0]);
dff10_chain II4 ( x[11], x[10], x[9:0], CK_coef, net89);
twos_comp10 II3 ( net72[0:9], CK, net138[0:9]);
dff10_in II2 ( net138[0:9], CK, net67[0:9]);
dff10_inX3 II1 ( net67[0:9], CK, In[9:0]);
ADD_BLOCK II0 ( net58[0:1], ACC_OUT[15:0], net66[0:9], net89,
               A3IN[1:0], { cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
               cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
               cds_globals.GND_, cds_globals.GND_, cds_globals.GND_);}
C.2.1 Enhanced Polynomial mapper

polynomial mapper

// Library - HDLib, Cell - full_opti2, View - schematic
'timescale ins / 10ps

module full_opti2 ( C0, C1, C2, C4, CK, IN );

input CK;
output [3:0] C0;
output [3:0] C1;
output [3:0] C2;
output [1:0] C4;

input [9:0] IN;

// Buses in the design
wire [0:3] net116;
wire [0:3] net112;
wire [0:3] net160;
wire [0:3] net106;
wire [0:3] net193;
wire [0:3] net100;
wire [0:3] net134;
wire [0:3] net81;
wire [0:3] net98;
wire [0:3] net56;
wire [0:3] net63;
wire [1:0] c;
wire [0:3] net103;
wire [0:3] net191;
wire [0:3] net158;
wire [0:3] net126;
wired [0:3] net74;
wire [0:3] net73;

specify
  specparam CDS_LIBNAME = "HDLib";
  specparam CDS_CELLNAME = "full_opti2";
  specparam CDS_VIEWNAME = "schematic";
endspecify

dff2_in I87 ( C4[1:0], CK, c[1:0]);
optmead I73 ( net81[0:3], net98[0:3], net79, CK, net193[0:3],
  cds Globals.GND_, net116[0:3]);
optmead I74 ( net74[0:3], net73[0:3], net72, CK, net63[0:3], net79,
  net112[0:3]);
splic10_3 I68 ( net160[0:3], net191[0:3], net158[0:3], IN[9:0]);
dff4_in I80 ( net56[0:3], CK, net100[0:3]);
dff4_in I76 ( net112[0:3], CK, net81[0:3]);
dff4_in I82 ( C1[3:0], CK, net103[0:3]);
dff4_in I77 ( net106[0:3], CK, net74[0:3]);
dff4_in I83 ( C0[1:0], CK, net56[0:3]);
dff4_in I86 ( net63[0:3], CK, net184[0:3]);
dff4_in I78 ( net101[0:3], CK, net73[0:3]);
dff4_in I70 ( net116[0:3], CK, net160[0:3]);
dff4_in I79 ( C2[3:0], CK, net26[0:3]);
dff4_in I53 ( net193[0:3], CK, net191[0:3]);
dff4_in I54 ( net191[0:3], CK, net159[0:3]);
dff4_in I56 ( net100[0:3], CK, net98[0:3]);
opti I75 ( net136[0:3], c[0], c[1], CK, net106[0:3]);
endmodule

optimapper

// Verilog HDL for "HDLib", "opti" "behavioral"

module opti (a, c0, c3, clk, in);
  output [3:0] a;
  output c0;
  output c3;
  input clk;
  input [3:0] in;
reg [5:0] r0mopt[0:15];
initial
  $readmemh("/home/vlsi/mshakha/CMOS35/HDLib/romoptbin.txt", r0mopt, 0.15);

  assign #1 (a,c3,c0)=(clk==1?r0mopt[in]:0);
endmodule

optimapper

// Library - HDLib, Cell - optmead, View - schematic
  `timescale ins / 10ps
module optmead ( S, a, cout, CK, b, cin, in );

output cout;
input CK, cin;
output [3:0] a;
output [3:0] S;
input [3:0] b;
input [3:0] in;

// Buses in the design
wire [0:3] net28;

specify
    specparam CDS_LIBNAME = "HDLib";
    specparam CDS_CELLNAME = "optmead";
    specparam CDS_VIEWNAME = "schematic";
endspecify

adder25 i50 ( cout, S[3:0], cin, net28[0:3], b[3:0], CK);
opti i47 ( a[3:0], net36, net10, CK, in[3:0]);
merge22 i59 ( net28[0:3], net36, net10, net10, net10);
endmodule

C.2.2 Evaluation map block

// Library - HDLib, Cell - IN_MAP_test, View - schematic
//timescale 1ps
module IN_MAP_test ( A0, A1, A2, A3, A4, CK, bb0, bb1, bb2 ),

input CK;
output [8:0] A1;
output [8:0] A2;
output [8:0] A0;
output [8:0] A3;
output [8:0] A4;
input [3:0] bb0;
input [3:0] bb1;
input [3:0] bb2;

// Buses in the design
wire [0:7] net449;
wire [0:7] net626;
wire [0:7] net870;
wire [0:7] net712;
wire [0:7] net718;
wire [7:0] coef23;
wire [7:0] C0d;
wire [7:0] b1i;
wire [7:0] b2i;
wire [7:0] b2id;
wire [8:8] b2dd;
wire [8:0] b2;
wire [8:0] b1;
wire [8:0] b0;
wire [0:7] net624;
wire [7:0] C2;
wire [7:0] C1;
wire [7:0] CO;
wire [7:0] coef14;
wire [7:0] blid;
wire [7:0] b2idd;
wire [7:0] coef13;
wire [8:0] b0d;
wire [7:0] coef24;
wire [0:7] net745;
wire [2:7] net700;
wire [0:7] net864;
wire [0:7] net543;
wire [0:7] net852;
wire [0:7] net754;
wire [0:7] net715;
wire [0:7] net724;
wire [0:7] net613;
wire [0:7] net742;
wire [0:8] net657;
wire [0:7] net611;
wire [0:7] net547;
wire [0:7] net748;
wire [0:7] net709;
wire [0:7] net730;
wire [0:7] net652;
wire [0:7] net703;
wire [0:7] net637;
wire [0:7] net650;
wire [0:7] net539;
wire [0:7] net588;
wire [0:7] net757;
wire [0:8] net660;

specify
specparam CDS_LIBNAME = "HDLib";
specparam CDS_CELLNAME = "IN_MAP_test";
specparam CDS_VIEWNAME = "schematic";
endspecify

dff_inX2 II25 ( net587, CK, b2dd(8));
dff_inX2 II134 ( net591, CK, b2dd(8));
dff_inX2 II168 ( net594, CK, bl(8));
dff_inX2 II104 ( net596, CK, bl(8));
dff_inX2 II146 ( net500, CK, b2(8));
dfmt2 II20 ( CK, coef13[7:0], blid[7:0], net613[0:7], b0d[7:0],
net611[0:7], cds_globals.VDD_, net612, NAIN, cds_globals.GND_,
net610, b0d[8], net609);
dfmt2 II137 ( CK, coef24[7:0], b2idd[7:0], net626[0:7], net745[0:7],
net624[0:7], net675, net625, net591, cds_globals.GND_, net623,
net688, A4[8]);
dfmt2 II132 ( CK, coef14[7:0], blid[7:0], net639[0:7], b0d[7:0],
net637[0:7], cds_globals.VDD_, net638, net681, cds_globals.GND_,
net636, b0d[8], net635);
dfmt2 II124 ( CK, coef23[7:0], b2idd[7:0], net652[0:7], net724[0:7],
net650[0:7], net672, net651, net587, cds_globals.GND_, net649,
module N_BLOCK ( Cout, out_coef, Ain0, CK, CK_coef, coef_in );

output out_coef;
input CK, CK_coef, coef_in;
output [8:0] Cout;
input [8:0] Ain0;

// Buses in the design
wire [0:7] net452;
wire [0:7] net720;
wire [0:7] net906;
wire [0:7] net382;
wire [0:7] net340;
wire [0:7] net636;
wire [0:7] net592;
wire [0:7] net33;
wire [0:7] net564;
wire [0:7] net804;
wire [0:7] net1014;
wire [0:7] net482;
wire [0:7] net454;
wire [0:7] net61;
wire [0:7] net990;
wire [0:7] net328;
wire [0:7] net44;
wire [0:7] net330;
wire [0:7] net438;
wire [0:7] net930;
wire [0:7] net202;
wire [0:7] net174;
wire [0:7] net846;
wire [0:7] net398;

C.2.3 Finite Field Computational Channel
"/ Library - HDLib, Cell - N_BLOCK, View - schematic
timescale ins : 10ps

module N_BLOCK ( Cout, out_coef, Ain0, CK, CK_coef, coef_in );

output out_coef;
input CK, CK_coef, coef_in;
output [8:0] Cout;
input [8:0] Ain0;

// Buses in the design
wire [0:7] net452;
wire [0:7] net720;
wire [0:7] net906;
wire [0:7] net382;
wire [0:7] net340;
wire [0:7] net636;
wire [0:7] net592;
wire [0:7] net33;
wire [0:7] net564;
wire [0:7] net804;
wire [0:7] net1014;
wire [0:7] net482;
wire [0:7] net454;
wire [0:7] net61;
wire [0:7] net990;
wire [0:7] net328;
wire [0:7] net44;
wire [0:7] net330;
wire [0:7] net438;
wire [0:7] net930;
wire [0:7] net202;
wire [0:7] net174;
wire [0:7] net846;
wire [0:7] net398;
wire [0:7] net900;
wire [0:7] net1008;
wire [0:7] net158;
wire [0:7] net648;
wire [7:0] a0;
wire [0:7] net1041;
wire [0:7] net186;
wire [0:7] net314;
wire [0:7] net718;
wire [0:7] net891;
wire [0:7] net1029;
wire [0:7] net909;
wire [0:7] net354;
wire [0:7] net860;
wire [0:7] net984;
wire [0:7] net272;
wire [0:7] net606;
wire [0:7] net242;
wire [0:7] net704;
wire [0:7] net746;
wire [0:7] net945;
wire [0:7] net963;
wire [0:7] net214;
wire [0:7] net566;
wire [0:7] net993;
wire [0:7] net1032;
wire [0:7] net370;
wire [0:7] net933;
wire [0:7] net978;
wire [0:7] net11;
wire [0:7] net578;
wire [0:7] net1044;
wire [0:7] net368;
wire [0:7] net480;
wire [0:7] net996;
wire [0:7] net244;
wire [0:7] net966;
wire [0:7] net762;
wire [0:7] net494;
wire [0:7] net228;
wire [0:7] net424;
wire [0:7] net160;
wire [0:7] net960;
wire [0:7] net748;
wire [0:7] net936;
wire [0:7] net969;
wire [0:7] net1023;
wire [0:7] net216;
wire [0:7] net975;
wire [0:7] net188;
wire [0:7] net662;
wire [0:7] net732;
wire [0:7] net790;
wire [0:7] net258;
wire [0:7] net1017;
wire [0:7] net1602;
wire [0:7] net298;
wire [0:7] net608;
wire [0:7] net496;
wire [0:7] net816;
wire [0:7] net760;
wire [0:7] net146;
wire [0:7] net342;
wire [0:7] net948;
wire [0:7] net664;
wire [0:7] net524;
wire [0:7] net1002;
wire [0:7] net774;
wire [0:7] net734;
wire [0:7] net286;
wire [0:7] net426;
wire [0:7] net123;
wire [0:7] net312;
wire [0:7] net552;
wire [0:7] net894;
wire [0:7] net832;
wire [0:7] net510;
wire [0:7] net1038;
wire [0:7] net300;
wire [0:7] net200;
wire [0:7] net412;

specify
  specparam CDS_LIBNAME = "HDLib";
  specparam CDS_CELLNAME = "N_BLOCK";
  specparam CDS_VIEWNAME = "schematic";
endspecify

adder8 I177 ( net1583, Cout[7:0]. net1501, net1602[0:7],
  (cdsGlobals.GND_, cdsGlobals.GND_, cdsGlobals.GND_,
  cdsGlobals.GND_, cdsGlobals.GND_, cdsGlobals.GND_,
  cdsGlobals.GND_, cdsGlobals.GND_), CK);
dff_chain I453 ( out_coef, net128[0:7], CK_coef, net141);
dff_inX2 I104 ( net132, CK, Ain0[8]);
dfmt_block I400 ( net146[0:7], net1603, net1602[0:7], net143, Cout[8],
  net141, net188[0:7], net1061, CK, CK_coef, net891[0:7], net1058,
  net1055, net183);
dfmt_block I399 ( net160[0:7], net159, net158[0:7], net157, net156,
  net155, net258[0:7], net1064, CK, CK_coef, net894[0:7], net1067,
  net1070, net253);
dfmt_block I398 ( net174[0:7], net173, net172[0:7], net171, net170,
  net169, net160[0:7], net1079, CK, CK_coef, net897[0:7], net1076,
  net1073, net155);
dfmt_block I397 ( net188[0:7], net187, net186[0:7], net185, net184,
  net183, net174[0:7], net1082, CK, CK_coef, net900[0:7], net1085,
  net1088, net169);
dfmt_block I396 ( net202[0:7], net201, net200[0:7], net199, net198,
  net197, net314[0:7], net1097, CK, CK_coef, net903[0:7], net1094,
  net1091, net309);
dfmt_block I395 ( net216[0:7], net215, net214[0:7], net213, net212,
  net211, net20210[7], net1100, CK, CK_coef, net906[0:7], net1103,
  net1106, net197);
dfmt_block I394 ( net230[0:7], net229, net228[0:7], net227, net226,
  net225, net216[0:7], net1115, CK, CK_coef, net909[0:7], net1112,
  net1109, net211);
dfmt_block I393 ( net244[0:7], net243, net242[0:7], net241, net240,
  net239, net230[0:7], net1118, CK, CK_coef, net912[0:7], net1121,
dfmt_block I231  ( net720[0:7], net719, net718[0:7], net717, net716, net715, net734[0:7], net1430, CK, CK_coef, net1014[0:7], net1427. net1424, net729);
dfmt_block I232  ( net706[0:7], net705, net704[0:7], net703, net702, net701, net692[0:7], net1415, CK, CK_coef, net1011[0:7], net1418, net1421, net687);
dfmt_block I233  ( net692[0:7], net691, net690[0:7], net689, net688, net687, net678[0:7], net1412, CK, CK_coef, net1008[0:7], net1409, net1406, net673);
dfmt_block I234  ( net678[0:7], net677, net676[0:7], net675, net674, net673, net664[0:7], net1397, CK, CK_coef, net1005[0:7], net1400, net1403, net659);
dfmt_block I235  ( net644[0:7], net643, net642[0:7], net641, net660, net659, net818[0:7], net1394, CK, CK_coef, net1002[0:7], net1391, net1388, net813);
dfmt_block I15  ( net33[0:7], net12, net31[0:7], net30, net29, net877, net46[0:7], net1523, CK, CK_coef, net11[0:7], net6, net19, net863);
dfmt_block I12  ( net46[0:7], net45, net44[0:7], net43, net42, net863, a0[7:0], cds_globals.VDD_, CK, CK_coef, (cds_globals.GND_,
                             cds_globals.GND, cds_globals.GND, cds_globals.GND, cds_globals.GND, cds_globals.GND,.,
                             cds_globals.GND, NAN_a0, cds_globals.VDD_, coef_in);
dff8_in I413  ( net891[0:7], CK, net186[0:7]);
dff8_in I412  ( net894[0:7], CK, net256[0:7]);
dff8_in I411  ( net897[0:7], CK, net158[0:7]);
dff8_in I410  ( net900[0:7], CK, net172[0:7]);
dff8_in I409  ( net903[0:7], CK, net312[0:7]);
dff8_in I408  ( net906[0:7], CK, net200[0:7]);
dff8_in I407  ( net509[0:7], CK, net214[0:7]);
dff8_in I406  ( net912[0:7], CK, net228[0:7]);
dff8_in I405  ( net915[0:7], CK, net270[0:7]);
dff8_in I404  ( net918[0:7], CK, net284[0:7]);
dff8_in I403  ( net921[0:7], CK, net298[0:7]);
dff8_in I402  ( net924[0:7], CK, net242[0:7]);
dff8_in I401  ( net927[0:7], CK, net592[0:7]);
dff8_in I327  ( net930[0:7], CK, net606[0:7]);
dff8_in I326  ( net933[0:7], CK, net326[0:7]);
dff8_in I325  ( net936[0:7], CK, net140[0:7]);
dff8_in I324  ( net939[0:7], CK, net354[0:7]);
dff8_in I323  ( net942[0:7], CK, net396[0:7]);
dff8_in I322  ( net945[0:7], CK, net410[0:7]);
dff8_in I321  ( net948[0:7], CK, net424[0:7]);
dff8_in I320  ( net951[0:7], CK, net368[0:7]);
dff8_in I319  ( net954[0:7], CK, net494[0:7]);
dff8_in I318  ( net957[0:7], CK, net438[0:7]);
dff8_in I317  ( net960[0:7], CK, net452[0:7]);
dff8_in I316  ( net963[0:7], CK, net466[0:7]);
dff8_in I315  ( net966[0:7], CK, net508[0:7]);
dff8_in I314  ( net969[0:7], CK, net522[0:7]);
dff8_in I313  ( net972[0:7], CK, net536[0:7]);
dff8_in I312  ( net975[0:7], CK, net382[0:7]);
dff8_in I311  ( net978[0:7], CK, net564[0:7]);
dff8_in I310  ( net981[0:7], CK, net578[0:7]);
dff8_in I309  ( net984[0:7], CK, net480[0:7]);
dff8_in I308  ( net987[0:7], CK, net550[0:7]);
dff8_in I284  ( net990[0:7], CK, net648[0:7]);
dff8_in I273  ( net993[0:7], CK, net718[0:7]);
dff8_in I272  ( net996[0:7], CK, net620[0:7]);
diff_in I385 (net1178, CK, net607);
diff_in I384 (net1181, CK, net327);
diff_in I383 (net1184, CK, net325);
diff_in I382 (net1187, CK, net324);
diff_in I381 (net1190, CK, net338);
diff_in I380 (net1193, CK, net339);
diff_in I379 (net1196, CK, net341);
diff_in I378 (net1199, CK, net355);
diff_in I377 (net1202, CK, net353);
diff_in I376 (net1205, CK, net352);
diff_in I375 (net1208, CK, net394);
diff_in I374 (net1211, CK, net395);
diff_in I373 (net1214, CK, net397);
diff_in I372 (net1217, CK, net411);
diff_in I371 (net1220, CK, net409);
diff_in I370 (net1223, CK, net408);
diff_in I369 (net1226, CK, net422);
diff_in I368 (net1229, CK, net423);
diff_in I367 (net1232, CK, net425);
diff_in I366 (net1235, CK, net369);
diff_in I365 (net1238, CK, net367);
diff_in I364 (net1241, CK, net366);
diff_in I363 (net1244, CK, net492);
diff_in I362 (net1247, CK, net493);
diff_in I361 (net1250, CK, net495);
diff_in I360 (net1253, CK, net439);
diff_in I359 (net1256, CK, net437);
diff_in I358 (net1259, CK, net436);
diff_in I357 (net1262, CK, net450);
diff_in I356 (net1265, CK, net451);
diff_in I355 (net1268, CK, net453);
diff_in I354 (net1271, CK, net467);
diff_in I353 (net1274, CK, net465);
diff_in I352 (net1277, CK, net464);
diff_in I351 (net1280, CK, net506);
diff_in I350 (net1283, CK, net507);
diff_in I349 (net1286, CK, net509);
diff_in I348 (net1289, CK, net523);
diff_in I347 (net1292, CK, net521);
diff_in I346 (net1295, CK, net520);
diff_in I345 (net1298, CK, net534);
diff_in I344 (net1301, CK, net535);
diff_in I343 (net1304, CK, net537);
diff_in I342 (net1307, CK, net383);
diff_in I341 (net1310, CK, net381);
diff_in I340 (net1313, CK, net380);
diff_in I339 (net1316, CK, net562);
diff_in I338 (net1319, CK, net563);
diff_in I337 (net1322, CK, net565);
diff_in I336 (net1325, CK, net579);
diff_in I335 (net1328, CK, net577);
diff_in I334 (net1331, CK, net576);
diff_in I333 (net1334, CK, net478);
diff_in I332 (net1337, CK, net479);
diff_in I331 (net1340, CK, net481);
diff_in I330 (net1343, CK, net551);
diff_in I329 (net1346, CK, net549);
diff_in I328 (net1349, CK, net548);
diff_in I287 (net1352, CK, net646);
C.2.4 Inverse MRRNS Map

Inverse Vandermonde matrix

```
// Library - HDLib, Cell - OUTMAP_COEF, View - schematic
// LAST TIME SAVED: May 13 15:32:18 1999
// NETLIST TIME: Jun 29 12:04:02 1999
`timescale lns / 10ps

module OUTMAP_COEF ( coef11, coef12, coef13, coef14, coef15, coef21,
         coef22, coef23, coef24, coef25, coef31, coef32, coef33, coef34,
         coef35, coef41, coef42, coef43, coef44, coef45, coef51, coef52,
         coef53, coef54, coef55 );

output [7:0] coef12;
output [7:0] coef32;
output [7:0] coef22;
output [7:0] coef42;
output [7:0] coef53;
output [7:0] coef13;
output [7:0] coef33;
output [7:0] coef23;
output [7:0] coef43;
output [7:0] coef14;
output [7:0] coef34;
output [7:0] coef24;
output [7:0] coef44;
output [7:0] coef55;
output [7:0] coef15;
output [7:0] coef35;
output [7:0] coef25;
output [7:0] coef45;
output [7:0] coef21;
output [7:0] coef41;
output [7:0] coef52;
output [7:0] coef51;
output [7:0] coef11;
output [7:0] coef31;

specparam CDS_LIBNAME = "HDLib";
specparam CDS_CELLNAME = "OUTMAP_COEF";
specparam CDS_VIEWNAME = "schematic";
```
endspecify

merge8 I89 ( coef35[7:0], cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.VDD_, cds_globals.VDD_, cds_globals.GND_ );
merge8 I88 ( coef45[7:0], cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.VDD_, cds_globals.VDD_, cds_globals.GND_ );
merge8 I87 ( coef15[7:0], cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.VDD_, cds_globals.VDD_, cds_globals.GND_ );
merge8 I86 ( coef25[7:0], cds_globals.VDD_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.VDD_, cds_globals.VDD_, cds_globals.GND_ );
merge8 I85 ( coef55[7:0], cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.VDD_, cds_globals.VDD_, cds_globals.GND_ );
merge8 I71 ( coef34[7:0], cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.VDD_, cds_globals.VDD_, cds_globals.GND_ );
merge8 I70 ( coef44[7:0], cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.VDD_, cds_globals.VDD_, cds_globals.GND_ );
merge8 I69 ( coef14[7:0], cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.VDD_, cds_globals.VDD_, cds_globals.GND_ );
merge8 I68 ( coef24[7:0], cds_globals.VDD_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.VDD_, cds_globals.VDD_, cds_globals.GND_ );
merge8 I67 ( coef54[7:0], cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.VDD_, cds_globals.VDD_, cds_globals.GND_ );
merge8 I53 ( coef33[7:0], cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.VDD_, cds_globals.VDD_, cds_globals.GND_ );
merge8 I52 ( coef43[7:0], cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.VDD_, cds_globals.VDD_, cds_globals.GND_ );
merge8 I51 ( coef13[7:0], cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.VDD_, cds_globals.VDD_, cds_globals.GND_ );
merge8 I50 ( coef23[7:0], cds_globals.VDD_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.VDD_, cds_globals.VDD_, cds_globals.GND_ );
merge8 I49 ( coef53[7:0], cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.VDD_, cds_globals.VDD_, cds_globals.GND_ );
merge8 I21 ( coef32[7:0], cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.VDD_, cds_globals.VDD_, cds_globals.GND_ );
merge8 I20 ( coef42[7:0], cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.VDD_, cds_globals.VDD_, cds_globals.GND_ );
merge8 I19 ( coef12[7:0], cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.VDD_, cds_globals.GND_, cds_globals.GND_ );
merge8 I18 ( coef22[7:0], cds_globals.VDD_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.VDD_, cds_globals.GND_ );
merge8 I17 ( coef52[7:0], cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.VDD_,
cds_globals.VDD_, cds_globals.VDD_, cds_globals.GND_);   
merge8 I13 ( coef51[7:0], cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.VDD_,
cds_globals.VDD_, cds_globals.VDD_, cds_globals.GND_);   
merge8 I11 ( coef51[7:0], cds_globals.VDD_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.VDD_,
cds_globals.VDD_, cds_globals.VDD_, cds_globals.GND_);   
merge8 I30 ( coef51[7:0], cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.VDD_,
cds_globals.VDD_, cds_globals.GND_, cds_globals.GND_);   
merge8 I32 ( coef51[7:0], cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.VDD_,
cds_globals.VDD_, cds_globals.GND_, cds_globals.GND_);   
merge8 I34 ( coef51[7:0], cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.VDD_,
cds_globals.VDD_, cds_globals.GND_);   
merge8 I36 ( coef51[7:0], cds_globals.GND_, cds_globals.GND_,
cds_globals.GND_, cds_globals.GND_, cds_globals.VDD_,
cds_globals.VDD_, cds_globals.GND_);   

endmodule

Inverse mapper

// Library - HFLib, Cell - OUT_MAP, View - schematic
// LAST TIME SAVED: Jun 29 12:03:28 1999
// NETLIST TIME: Jun 29 12:04:03 1999
'timescale ins : 10ps

module OUT_MAP ( OUT0, OUT1, OUT2, OUT3, OUT4, CK, In0, In1, In2, In3,
In4 );

input CK;
output [7:0] OUT2;
output [7:0] OUT4;
output [7:0] OUT3;
output [7:0] OUT1;
output [7:0] OUT0;
input [8:0] In0;
input [8:0] In2;
input [8:0] In3;
input [8:0] In4;
input [8:0] In1;

// Buses in the design
wire [0:7] net684;
wire [0:7] net956;
wire [0:7] net680;
wire [0:7] net705;
wire [0:7] net589;
wire [0:7] net826;
wire [0:7] net824;
wire [0:7] net852;
wire [0:7] net604;
wire [0:7] net746;
wire [0:7] net601;
wire [0:7] net616;
wire [0:7] net761;
wire [0:7] net982;
wire [0:7] net837;
wire [0:7] net697;
wire [0:7] net595;
wire [0:7] net693;
wire [7:0] coef11;
wire [7:0] coef22;
wire [7:0] coef25;
wire [7:0] coef52;
wire [7:0] coef53;
wire [7:0] coef54;
wire [7:0] coef55;
wire [7:0] coef51;
wire [7:0] coef42;
wire [7:0] coef43;
wire [7:0] coef44;
wire [7:0] coef45;
wire [0:7] net562;
wire [7:0] coef41;
wire [7:0] coef32;
wire [7:0] coef33;
wire [7:0] coef34;
wire [7:0] coef12;
wire [7:0] coef13;
wire [7:0] coef14;
wire [0:7] net619;
wire [7:0] coef15;
wire [7:0] coef35;
wire [7:0] coef31;
wire [7:0] a0;
wire [7:0] a1;
wire [7:0] coef23;
wire [7:0] coef21;
wire [7:0] a3;
wire [7:0] coef24;
wire [0:7] net1197;
wire [7:0] a2;
wire [7:0] a4;
wire [0:7] net800;
wire [0:7] net839;
wire [0:7] net850;
wire [0:7] net687;
wire [0:7] net980;
wire [0:7] net878;
wire [0:7] net696;
wire [0:7] net652;
wire [0:7] net813;
wire [0:7] net930;
wire [0:7] net669;
wire [0:7] net580;
wire [0:7] net865;
wire [0:7] net1199;
wire [0:7] net1195;
wire [0:7] net681;
wire [0:7] net678;
wire [0:7] net995;
wire [0:7] net993;
wire [0:7] net946;
wire [0:7] net583;
wire [0:7] net665;
wire [0:7] net625;
wire [0:7] net748;
wire [0:7] net641;
wire [0:7] net928;
wire [0:7] net876;
wire [0:7] net607;
wire [0:7] net969;
wire [0:7] net691;
wire [0:7] net568;
wire [0:7] net737;
wire [0:7] net622;
wire [0:7] net1193;
wire [0:7] net685;
wire [0:7] net699;
wire [0:7] net592;
wire [0:7] net708;
wire [0:7] net915;
wire [0:7] net702;
wire [0:7] net798;
wire [0:7] net904;
wire [0:7] net774;
wire [0:7] net639;
wire [0:7] net610;
wire [0:7] net917;
wire [0:7] net598;
wire [0:7] net902;
wire [0:7] net967;
wire [0:7] net889;
wire [0:7] net577;
wire [0:7] net654;
wire [0:7] net937;
wire [0:7] net811;
wire [0:7] net863;
wire [0:7] net934;
wire [0:7] net759;
wire [0:7] net633;
wire [0:7] net772;
wire [0:7] net943;
wire [0:7] net941;
wire [0:7] net667;
wire [0:7] net571;
wire [0:7] net1194;
wire [0:7] net954;
wire [0:7] net785;
wire [0:7] net857;
wire [0:7] net672;
wire [0:7] net891;

specparam CDS_LIBNAME = "HDLib";
specparam CDS_CELLNAME = "OUT_MAP";
specparam CDS_VIEWNAME = "schematic";
endspecify

type adder8 I198 ( net1328, OUT[7:0], net673, net697[0:7],

290
```c
dff_inX2 I52 ( net519, CX, In4[8]);
dff_inX2 I50 ( net522, CX, In1[8]);
dff_inX2 I48 ( net525, CX, In2[8]);
dff_inX2 I46 ( net528, CX, In1[8]);
dff_inX2 I104 ( net531, CX, In0[8]);
outmap_Coef I44 ( coef11[7:0], coef12[7:0], coef13[7:0], coef14[7:0],
                 coef15[7:0], coef21[7:0], coef22[7:0], coef23[7:0], coef24[7:0],
                 coef25[7:0], coef31[7:0], coef32[7:0], coef33[7:0], coef34[7:0],
                 coef35[7:0], coef41[7:0], coef42[7:0], coef43[7:0], coef44[7:0],
                 coef45[7:0], coef51[7:0], coef52[7:0], coef53[7:0], coef54[7:0],
                 coef55[7:0]);
dff8_inX2 I43 ( a4[7:0], CK, net562[0:7]);
dff8_inX2 I42 ( net562[0:7], CK, net563[0:7]);
dff8_inX2 I41 ( a3[7:0], CK, net571[0:7]);
dff8_inX2 I40 ( net568[0:7], CK, net577[0:7]);
dff8_inX2 I39 ( net571[0:7], CK, net580[0:7]);
dff8_inX2 I38 ( a2[7:0], CK, net583[0:7]);
dff8_inX2 I37 ( net577[0:7], CK, net562[0:7]);
dff8_inX2 I36 ( net580[0:7], CK, net619[0:7]);
dff8_inX2 I35 ( net583[0:7], CK, net616[0:7]);
dff8_inX2 I78 ( a1[7:0], CK, net625[0:7]);
dim_index_map I29 ( net589[0:7], CK, In2[7:0]);
dim_index_map I27 ( net592[0:7], CK, In1[7:0]);
dim_index_map I31 ( net595[0:7], CK, In3[7:0]);
dim_index_map I33 ( net598[0:7], CK, In4[7:0]);
dim_index_map I10 ( net601[0:7], CK, In0[7:0]);
dff8_in I119 ( net934[0:7], CK, net987[0:7]);
dff8_in I115 ( net937[0:7], CK, net837[0:7]);
dff8_in I113 ( net885[0:7], CK, net902[0:7]);
dff8_in I119 ( net971[0:7], CK, net665[0:7]);
dff8_in I117 ( net9461[0:7], CK, net772[0:7]);
dff8_in I173 ( net604[0:7], CK, net639[0:5]);
dff8_in I176 ( net607[0:7], CK, net678[0:7]);
dff8_in I177 ( net610[0:7], CK, net691[0:7]);
dff8_in I196 ( net708[0:7], CK, net993[0:7]);
dff8_in I1100 ( net699[0:7], CK, net980[0:7]);
dff8_in I199 ( net702[0:7], CK, net941[0:7]);
dff8_in I1126 ( net696[0:7], CK, net928[0:7]);
dff8_in I1127 ( net1199[0:7], CK, net889[0:7]);
dff8_in I1128 ( net1194[0:7], CK, net876[0:7]);
dff8_in I1129 ( net687[0:7], CK, net915[0:7]);
dff8_in I1162 ( net633[0:7], CK, net652[0:7]);
dff8_in I1151 ( net684[0:7], CK, net863[0:7]);
dff8_in I1152 ( net681[0:7], CK, net824[0:7]);
dff8_in I1153 ( net1195[0:7], CK, net811[0:7]);
dff8_in I1155 ( net857[0:7], CK, net850[0:7]);
dff8_in I1179 ( net672[0:7], CK, net798[0:7]);
dff8_in I1180 ( net669[0:7], CK, net759[0:7]);
dff8_in I1181 ( net1197[0:7], CK, net746[0:7]);
dff8_in I1182 ( net1193[0:7], CK, net785[0:7]);
dff8_in I130 ( net616[0:7], CK, net589[0:7]);
dff8_in I132 ( net169[0:7], CK, net595[0:7]);
dff8_in I134 ( net622[0:7], CK, net598[0:7]);
dff8_in I128 ( net625[0:7], CK, net392[0:7]);
dff8_in I1 ( a0[7:0], CK, net601[0:7]);
dff8_in I17 ( net705[0:7], CK, net954[0:7]);
dfmt2 I105 ( CK, coef22[7:0], a1[7:0], net956[0:7], net702[0:7],
             net954[0:7], cds_globals.GND_, net955, NAN_al, cds_globals.GND_,
             net953, net511, net952);
C.2.5 Binary Computational Channel

Pipeline adder (single stage)

// Library - HDLib, Cell - PIPE_ADD, View - schematic
// LAST TIME SAVED: Jun 30 11:52:18 1999
// NETLIST TIME: Jun 30 11:52:41 1999
	timescale ins / 10ps

module PIPE_ADD ( A3OUT, ACC_OUT, AQOUT, B3, A3, A, ACC_IN, B_in, CK, 
				CK_coef );

output B3;
input B_in, CK, CK_coef;
output [1:0] A3OUT;
output [9:0] AQOUT;
output [15:0] ACC_OUT;
input [9:0] A;
input [15:0] ACC_IN;
input [1:0] A3;

// Buses in the design
wire [9:0] An;
wire [9:0] bn;
wire [9:0] btc;
wire [9:0] b;
wire [9:0] Atc;
wire [1:0] A3B3;
wire [15:0] ACC;
wire [9:0] ax;
wire [9:0] bx;
wire [0:9] net265;
wire [0:7] net218;
wire [0:7] net122;
wire [0:7] net125;
wire [0:7] net212;
wire [0:7] net114;
wire [0:7] net117;
wire [0:1] net330;
wire [0:9] net140;
wire [0:7] net142;
wire [0:9] net257;
wire [0:9] net341;
wire [0:7] net123;
wire [0:7] net111;

specify
    specparam CDS_LIBNAME = "HDLib";
    specparam CDS_CELLNAME = "PIPE_ADD";
    specparam CDS_VIENNAME = "schematic";
endspecify

dff2_in I117 ( A3OUT[1:0], CK, net330[0:1]);
dff8_in I108 ( net122[0:7], CK, A3B3[1], A3B3[1], A3B3[1], A3B3[1], A3B3[1], A3B3[1], A3B3[1], A3B3[0], cds Globals.GND_);
dff8_in I111 ( ACC_OUT[15:8], CK, net117[0:7]);
dff8_in I112 ( ACC_OUT[7:0], CK, net114[0:7]);
dff8_in I109 ( net219[0:7], CK, net111[0:7]);
dff8_in I110 ( net125[0:7], CK, net123[0:7]);
mux10 I101 ( Acc[9:0], net140[0:9], A[9:0], A[9], net342, CK);
mux10 I102 ( bcc[9:0], net143[0:9], b[9:0], b[9], net132, CK);
twos_comp10 I97 ( net140[0:9], CK, A[9:0]);
twos_comp10 I98 ( net143[0:9], CK, b[9:0]);
twos_comp10 I95 ( bn[9:0], CK, bcc[9:0]);
twos_comp10 I96 ( An[9:0], CK, Acc[9:0]);
dff2_inX3 I94 ( net330[0:1], CK, A3[1:0]);
dff10_inX3 I93 ( net341[0:9], CK, A[9:0]);
dff16_inX3 I90 ( ACC[15:0], CK, ACC_IN[15:0]);
dff10_in I116 ( AGOUT[9:0], CK, net341[0:9]);
dff10_in I78 ( ax[9:0], CK, net257[0:9]);
dff10_in I77 ( bx[9:0], CK, net265[0:9]);
mux2_3 I60 ( A3B3[1:0], (cds Globals.GND_, cds Globals.GND_),(cds Globals.VDD_, cdss Globals.VDD_),(cds Globals.VDD_, cdss Globals.VDD_),(cds Globals.VDD_, ZC1, ZC2, ZC3, CK));
wor2_1 I53 ( ZC1, ZA1, ZB1);
wor2_1 I54 ( ZC2, net193, net187);
wor2_1 I55 ( ZC3, net184, net190);
wand2_1 I58 ( net184, ZA3, ZB2);
wand2_1 I57 ( net187, ZA3, ZB3);
wand2_1 I59 ( net190, ZA2, ZB3);
wand2_1 I56 ( net193, ZA2, ZB2);
wand2_1 I43 ( ZA3, A3[1], A3[0]);
wand2_1 I42 ( ZA2, net246, A3[0]);
wand2_1 I19 ( ZB3, B3, B0);
wand2_1 I22 ( ZB2, net248, B0);
wnor2_1 I44 ( ZA1, A3[1], A3[0]);
C.3 Test benches

C.3.1 fullopti

// timescale set according to user specification in the STL defiming statement

`timescale lns : 10ps

module test;

reg CK;
wire [3:0] C1;
wire [3:0] C4;
wire [3:0] C0;
wire [3:0] C2;
reg [9:0] IN;

full_opti top(C0, C1, C2, C4, CK, IN);

// parameter Z = 1'b0;
initial
begin
CK = 0;
IN[9:0] = 0;
end
end
// 10 Verilog time points generated

initial #12000 $stop;
always #40 CK = ~ CK;
always @(negedge CK) #5 IN=IN+1;
endmodule

C.3.2 fullopti2

// timescale set according to user specification in the STL deftiming statement

`timescale lns / 10ps
module test;

reg CK;
wire [3:0] C1;
wire [3:0] C4;
wire [3:0] C0;
wire [3:0] C2;
reg [9:0] IN;

full_opti2 top(C0, C1, C2, C4, CK, IN);

//parameter Z = l'bz;
initial
begin
  CK = 0;
  IN[9:0] =0;
end
// 10 Verilog time points generated

initial #12000 $stop;
always #40 CK = ~ CK;
always @(negedge CK) #5 IN=IN+1;
endmodule

C.3.3 Add_stage

`timescale lns / 10ps

module test;

reg CK;
wire [9:0] S;
reg [9:0] In;

ADD_STAGE top(S, CK, In);

//parameter Z = l'bz;
initial
begin
CK = 0;
In[9:0] = 0;
end

// 10 Verilog time points generated

initial #30000 $stop;
always #40 CK = ~ CK;
always @(posedge CK) #5 In=In+1;
endmodule

C.3.4 Fermat ALU Test

Original Fermat block

// Library - HDLib, Cell - fermat_test2, View - schematic
// LAST TIME SAVED: Apr 19 10:49:20 1999
// NETLIST TIME: Apr 19 10:50:19 1999
`timescale 1ns / 10ps

module fermat_test2 ( cout, Ain, CK, Cin, IN_COEF );

input CK;
output [8:0] cout;
input [7:0] Cin;
input [7:0] IN_COEF;
input [7:0] Ain;

// Buses in the design

wire [0:7] net35;
wire [8:0] coef;
wire [0:7] net38;
wire [0:7] net22;
wire [8:0] a;
wire [8:0] c;

specify
    specparam CDS_LIBNAME = "HDLib";
    specparam CDS_CELLNAME = "fermat_test2";
    specparam CDS_VIEWNAME = "schematic";
endspecify

dim_index_map I4 ( net38[0:7], CK, coef[7:0] );
dim_index_map I5 ( net35[0:7], CK, a[7:0] );
diminished_1 I2 ( a[8:0], Ain[7:0] );
diminished_1 I1 ( coef[8:0], IN_COEF[7:0] );
diminished_1 I3 ( c[8:0], Cin[7:0] );
dfmt257 I0 ( CK, net38[0:7], net35[0:7], net22[0:7], c[7:0], cout[7:0],
    c[8], cout[8], a[8], coef[8], net19 );
endmodule
Original Fermat ALU Test

`timescale 1ns / 10ps

module test;

reg CK;
wire [8:0] cout;
reg [7:0] Cin;
reg [7:0] Ain;
reg [7:0] IN_COEF;

fermat_test2 top(cout, Ain, CK, Cin, IN_COEF);

// Verilog stimulus file.
// Please do not create a module in this file.

// Default verilog stimulus.

initial
begin
    CK = 0;
    Ain[7:0] =0;
    Cin[7:0] =0;
    IN_COEF[7:0] =1;
end
// 10 Verilog time points generated

initial #12000 $stop;

always @ CK = ~ CK;
always @(negedge CK) @5 Ain=Ain+1;

endmodule

Enhanced Fermat ALU block

// Library - HDLib, Cell - fermat_test, View - schematic
// LAST TIME SAVED: Apr 19 11:50:11 1999
// NETLIST TIME: Apr 19 11:50:34 1999
`timescale 1ns / 10ps

module fermat_test ( C01, aout[7:0], aout[8], cout[7:0], cout[8],
    Ain[7:0], C11, CK, Cin[7:0], IN_COEF[7:0] );

output C01;
input C11, CK;
output [8:0] aout;
output [8:0] cout;
input [7:0] Ain;
input [7:0] IN_COEF;
input [7:0] Cin;
// Buses in the design
wire [0:7] net95;
wire [8:0] coeff;
wires [8:0] a;
wires [8:0] b;
wires [0:7] net33;
wires [0:7] net15;
wires [0:8] net7;
wires [0:7] net31;

specify
    specparam CDS_LIBNAME = "HDLib";
    specparam CDS_CELLNAME = "fermat_test";
    specparam CDS_VIEWNAME = "schematic";
endspecify

dff_l17 I17 ( aout[7:0], CK, net95[0:7]);
dff_l1 I1 ( net33[0:7], CK, net1[0:7]);
dff_l16 I16 ( net13, CK, net28);
dff_l15 I15 ( net28, CK, a[8]);
dff_l14 I14 ( c[8:0], CK, net7[0:8]);
dim_index_map I12 ( net31[0:7], CK, a[7:0]);
dim_index_map I11 ( net15[0:7], CK, coeff[7:0]);
diminished_l13 I13 ( coeff[8:0], IN_COEF[7:0]);
diminished_l9 I9 ( net7[0:8], Cin[7:0]);
diminished_l7 I7 ( a[9:0], Ain[7:0]);
dfmt I5 ( CK, net15[0:7], net33[0:7], net95[0:7], c[7:0], cout[7:0],
    C1I, C0I, net14, coeff[8], aout[8], c[8], cout[8]);

endmodule

Enhanced Fermat ALUTest

' timescale 1ns / 10ps

module test;

wire COI;
reg CI, CK;
wires [8:0] aout;
wires [8:0] cout;
reg [7:0] Ain;
reg [7:0] IN_COEF;
reg [7:0] Cin;

fermat_test top(COI, aout[7:0], aout[8], cout[7:0], cout[8], Ain[7:0], CI, CK,
    Cin[7:0], IN_COEF[7:0]);

// Verilog stimulus file.
// Please do not create a module in this file.
// Default verilog stimulus.

initial begin
  CK = 0;
  CIi=0;
  Ain[7:0] =0;
  Cin[7:0] =0;
  IN_COEF[7:0] =1;
end

// 10 Verilog time points generated

initial $t0000 $stop;

always #40 CK = ~ CK;
always @(negedge CK) $5 Ain=Ain+1;

endmodule

Figure C.1 Fermat ALU Verilog Simulation

C.3.5 Filter Test
	'timescale ins / 10ps

module test;

wire out_coef;
reg CK, CK_coef, coef_in;
reg LOAD_DONE;
wire [15:0] ACC_OUT;
wire [7:0] OUT3;
wire [7:0] OUT4;
wire [7:0] OUT0;
wire [7:0] OUT1;
wire [7:0] OUT2;
reg [9:0] In;
// reg [0:11] Table[1:54];
// reg [0:9] Table2[1:318];
reg [3032:0] templ;
reg [3032:0] temp2;
reg [11:0] counter;

FILTER top(ACC_OUT, OUT0, OUT1, OUT2, OUT3, OUT4, out_coef, CK, CK_coef, In, 
  coef_in);

initial
  begin
    // $readmemh("/home/vlsi/mshahka/CMOS35/VER/strm.txt", Table1, 1, 54);
    // $readmemh("/home/vlsi/mshahka/CMOS35/VER/strm.txt", Table2, 35, 32);
    temp=Table[0];
    CK = 1'b0;
    CK_coef = 1'b0;
    In[9:0] = 10'b0000000000;
    coef_in = 1'b0;
  end
always
  begin
    wait (~LOAD_DONE)
    #40 CK_coef=-CK_coef;
    if (LOAD_DONE==1) CK_coef=1;
  end
always @(negedge CK_coef)
  begin
    if (LOAD_DONE==0)
      begin
        coef_in=temp[counter];
        counter=counter+1;
        // #40 CK_coef=-CK_coef;
      end
    //else
    // CK_coef=1;
    if (out_coef) LOAD_DONE=1;
  end
always
  begin
    wait (LOAD_DONE)
    #40 CK=-CK;
  end
always @(negedge CK) #5 In=In+1;
endmodule

C.3.6 Input map test
// timescale set according to user specification in the STL deftiming statement

`timescale lns / 10ps

module test;
reg CK;
wire [8:0] A2;
wire [8:0] A0;
wire [8:0] A1;
wire [7:0] A5;
wire [8:0] A3;
wire [8:0] A4;
reg [9:0] IN;

IN_MAP2 top(A0, A1, A2, A3, A4, A5, CK, IN);

//parameter Z = 1'b0;
initial
begin
    CK = 0;
    IN[9:0] =0;
end
// 10 Verilog time points generated
initial #12000 $stop;
always #40  CK = ~ CK;
always @(negedge CK) #5  IN=IN+1;
endmodule

Figure C.2 Input Mapper Schematic
C.3.7 Output Map test

```
`timescale 1ns / 10ps

module test;

reg CK;
wire [7:0] OUT1;
wire [7:0] OUT4;
wire [7:0] OUT2;
wire [7:0] OUT3;
reg [8:0] In0;
reg [8:0] In1;
reg [8:0] In2;
reg [8:0] In3;
reg [8:0] In4;

OUT_MAP top(OUT0, OUT1, OUT2, OUT3, OUT4, CK, In0, In1, In2, In3, In4);

initial
begin
    CK = 1'b0;
    In0[8:0] = 9'b000000000;
    In1[8:0] = 9'b000000000;
    In2[8:0] = 9'b000000000;
    In3[8:0] = 9'b000000000;
    In4[8:0] = 9'b000000000;
end
initial #24000 $stop;
always #40 CK = ~ CK;
```
always @(negedge CK) #5
begin
    In0=In0+1;
    In1=In1+1;
    In2=In2+1;
    In3=In3+1;
    In4=In4+1;
end
endmodule

Figure C.4 Output Mapper Schematic

Figure C.5 Output Mapper Layout

<table>
<thead>
<tr>
<th>est.CK</th>
<th>00b</th>
<th>00c</th>
<th>00d</th>
<th>00e</th>
<th>00f</th>
<th>010</th>
<th>011</th>
<th>012</th>
<th>013</th>
<th>014</th>
</tr>
</thead>
<tbody>
<tr>
<td>st.In0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t.OUT0</td>
<td>00</td>
<td>XX</td>
<td>XX</td>
<td>00</td>
<td>05</td>
<td>00</td>
<td>0a</td>
<td>00</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>t.OUT1</td>
<td>00</td>
<td>XX</td>
<td>XX</td>
<td>00</td>
<td>05</td>
<td>00</td>
<td>0a</td>
<td>00</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>t.OUT2</td>
<td>00</td>
<td>XX</td>
<td>XX</td>
<td>00</td>
<td>05</td>
<td>00</td>
<td>0a</td>
<td>00</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>t.OUT3</td>
<td>00</td>
<td>XX</td>
<td>XX</td>
<td>00</td>
<td>05</td>
<td>00</td>
<td>0a</td>
<td>00</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>t.OUT4</td>
<td>00</td>
<td>XX</td>
<td>XX</td>
<td>00</td>
<td>05</td>
<td>00</td>
<td>0a</td>
<td>00</td>
<td>01</td>
<td>00</td>
</tr>
</tbody>
</table>
C.3.8 Pipeline Adder test (binary channel)

`timescale 1ns / 10ps

module test;

wire B3;
wire BZ;
reg LOAD_DONE;
reg B_in, CK, CK_coef;
wire [15:0] ACC_OUT;
wire [1:0] A3OUT;
wire [9:0] AGOUT;
reg [9:0] A;
reg [15:0] ACC_IN;
reg [1:0] A3;
reg [12:0] Table[0:1];
reg [12:0] temp;
reg [3:0] counter;

PIPE_ADD top(A3OUT, ACC_OUT, AGOUT, B3, A3, A, ACC_IN, B_in, CK, CK_coef);
dff_in dummy(BZ, CK_coef, B3);

initial begin
 $readmemb("/home/visi.mshahka/CNOS35/VER coef str.txt", Table);
temp=Table[0];
  counter=0;
  CK = 0;
  CK_coef=0;
  A[9:0]=0;
  B_in=0;
  A3[1:0]=1;
  LOAD_DONE=0;
  ACC_IN[15:0]=0;
end

always begin
  wait (~LOAD_DONE)
  #40 CK_coef=-CK_coef;
  if (LOAD_DONE==1) CK_coef=1;
end

always @(negedge CK_coef)
begin
  if (LOAD_DONE==0)
  begin
    B_in=temp[counter];
    counter=counter+1;
    // #40 CK_coef=-CK_coef;
  end
  //else
    //CK_coef=1;
  if (B3) LOAD_DONE=1;
end
always begin
    wait (LOAD_DONE)
    #50 CK=¬CK;
end
always @(negedge CK) #5
begin
    A=A+1;
    A3=¬A3;
end

endmodule

Figure C.6 Pipeline Adder Schematic
C.4 C-files

C.4.1 binary_conv.c
#include <stdio.h>
#include <stdlib.h>

void streamout(int value, int length)
{
    int i;
    int a=value;

    if (a<0) a=(1 << length) - a;

    for (i=0;i<length;i++)
    {
        printf("%c", ( a & ( 1 << (length-i) ) ) == 0 ? '0' : '1' );

        a = a << 1;
    }

    fflush(stdout);
}

void main(void)
{
    streamout(10,9);

    return;
}
C.4.2 coeff.c
/* C include files */

#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/time.h>
#define TAP 53
#define IX 8
#define GEN 3
#define MOD 257

void streamout(int value, int length, FILE *str);

int coeff[TAP]={480,320,30,-96,-28,48,24,-24,-20,12,16,-3,-12,-3,9,7,-4,-8,0,8,2,-7,
-3,5,2,-3,-2,-3,2,5,-3,-7,2,8,0,-8,-4,7,8,-3,-12,-3,16,12,-20,-24,24,48,-28,-
96,30,320,480};

int pg[3][TAP];
int pc[4][TAP];

int i, j, k, l, counter, count[15];
int temp, temp2, templ;
int in_vec[5][TAP], out_vec[5][TAP];
int bin[MOD], idx[MOD];
int pwz;
FILE *fpo;

main()
{
  /* convert the filter coefficient to simple polymap with three coeffs */

  for(i=0; i<TAP; i++){
    temp2=coeff[i]%IX*IX);
    pg[2][i]=(coeff[i]-temp2)/(IX*IX);
    templ=temp2%IX;
    pg[1][i]=(temp2-templ)/IX;
    pg[0][i]=templ;
  }

  /*initialize counters for calculating pgfs */

  counter=0;
  for(i=0; i<15; i++)
    count[i]=0;

  /* initialize new polymap coeffs to zero */

  for(i=0;i<TAP;i++){
    pc[0][i]=0;
    pc[1][i]=0;
    pc[2][i]=0;
    pc[3][i]=0;
    counter=counter+1;
  }
}
/* new polymap coeffs */

for(i=0; i<3; i++)
  for(j=0; j<TAP; j++)
    if (pg[i][j]+pc[i][j])>4)
      pc[i][j]=pc[i][j]+pg[i][j]-8;
      pc[i+1][j]=pc[i+1][j]+1;
    else
      if(pg[i][j]+pc[i][j])<4)
        pc[i][j]=pc[i][j]+pg[i][j]+8;
        pc[i+1][j]=pc[i+1][j]-1;
    else
      pc[i][j]=pg[i][j]+pc[i][j];

/* for(i=0; i<TAP; i++)

printf("orig= %i %i %i  conv= %i %i %i \n", pg[2][i], pg[1][i], pg[0][i], pc[3][i], pc[2][i], pc[1][i], pc[0][i]); * /

/* input vector to the evaluation map */

for(i=0; i<TAP; i++)
  for(j=0; j<S; j++)
    in_vec[j][i]=0;

for(i=0; i<TAP; i++)
  in_vec[0][i]=pc[0][i];
  in_vec[1][i]=pc[1][i];
  in_vec[2][i]=pc[2][i];

/* output vector from evaluation map, all mod 257, no negative numbers*/

for(i=0; i<TAP; i++)
  out_vec[0][i]=in_vec[0][i]%MOD;
  out_vec[1][i]=(in_vec[0][i]+in_vec[1][i])%MOD;
  out_vec[2][i]=(in_vec[0][i]-in_vec[1][i])%MOD;
  out_vec[3][i]=(in_vec[0][i]+2*in_vec[1][i]+4*in_vec[2][i])%MOD;
  out_vec[4][i]=(in_vec[0][i]-2*in_vec[1][i]+4*in_vec[2][i])%MOD;

/* then find the index rep for all the out vecs*/
/* this routine takes the index and gives the number */

pwr=GEN;
bin[0]=1;
for (i=1; i<MOD-1; i++)
  bin[i]=bin[i-1]*pwr%MOD;

/* if(bin[i]<0)
    bin[i]=bin[i]+MOD; */
/* pwr=bin[i]*pwr; */
printf("bin[%i] = %i \n", i, bin[i]);

for( i=0; i<MOD-1; i++)
  inidx(bin[i])=i;
} /* reverse the above table, given number, give index */

    for(i=0; i<=MOD-1; i++)
        printf("indx[%d]= %d \n", i, indx[i]);

/* out_vec should be converted to the value of index should be represented as a 9bit
number, with the MSB signifying \nNAN. The only number with the MSB=1 is 0, which has no
index a string should be made in the form indx[][LSB ... MSB] */

/* also each each member of coeff[], should be represeted as 10bit 2's complement num-
ber. pc[3][] should also be converted to 2bit 2's comp. these tow should be concate-
nated to form an string, coeff[][LSB ... MSB]pc[3][][LSB MSB] */

/*the final serial loader string will be of the form:
outvec[0][52][LSB ... MSB] ... outvec[0][0]outvec[1][52][LSB ... MSB] ... out-
vec[1][0][LSB ... MSB]outvec[2][52][L ... M] ... outvec[2][0][L ... M]outvec[3][52][L ...
M] ... outvec[3][0][L ... M]outvec[4][52][L ... M] ... outvec[4][0][L ... M]
coeff[52][LSB ... MSB]pc[3][52][LSB MSB] ... coeff[0][LSB ... MSB]pc[3][0][LSB MSB]+(12
bit check flag) */

fpo=fopen ("strm.txt", "w");

/*use streamout to generate string, must be in reverse order for verilog */

streamout(1223, 12, fpo);
streamout(1223, 12, fpo);
streamout(1223, 12, fpo);
for(i=0; i<53; i++)
    streamout(pc[3][i], 12, fpo);
    fprintf(fpo, "\n");
}
for(i=0; i<53; i++)
    streamout(coeff[i], 10, fpo);
    fprintf(fpo, "\n");
}
for(i=4; i>=0; i--)
    for(j=0; j<53; j++)
        streamout(indx[out_vec[i][j]], 10, fpo);
        fprintf(fpo, "\n");
}
fclose(fpo);

void streamout(int value, int length, FILE *str)
{
    int i;
    int a=value;
    if (a<0) a=(1 << length) + a;
    for (i=0;i<length;i++)
    {
        fprintf(str,"%c", ( a & ( 1 << (length-1) ) ) == 0 ? '0' : '1' );
        a = a << 1;
    }
}
C.4.3 indmap.c

#include <stdio.h>
#include <math.h>
#define GEN 3
#define MOD 257

main()
{
    int i;
    int bin[256];
    int pwr;
    int number;

    FILE *fpo;

    fpo=fopen ("myind.txt", "w");
    pwr=GEN;
    bin[0]=GEN;
    for (i=1; i<=MOD-1; i++)
    {
        bin[i]=bin[i-1]*pwr%MOD;
        if(bin[i]<0)
            bin[i]=bin[i]+MOD; /*
        "pwr=bin[i]*pwr;"*/
    }

    for(number=0; number<=MOD-2; number++)
    {
        printf(fpo,"%d", number+1);
        /* printf(fpo, "\t"); */
        printf(fpo," %d", bin[number]);

        printf(fpo,"\n");
        printf(fpo,"%d", number+1);
        /* printf(fpo, "\t"); */
        printf(fpo," %d", bin[number]);

        printf(fpo,"\n");
    }
    fclose(fpo);
}

C.4.4 r256GenBin.c

#include <stdio.h>
#include <math.h>
main()
{
    int i, ind1, ind2, code1, code2, code[256];
    int bin[256][8];
    int pwr, remainder, weight;
    int number, line;

    FILE *fpi, *fpo;

    printf("start..... \n\n");

    /*open input file, read the data to an array*/
    fpi=fopen("revrom.txt", "r");

    for(i=0; i<=255; i++)
    {
        fscanf(fpi, "%d%d \n", &ind1, &code1);
        fscanf(fpi, "%d%d \n", &ind2, &code2);
        printf("%5d%5d\n", ind1, code1);
        if (code1!=code2)
        {
            printf("%5d%5d.....an error occurred!\n", ind2, ind2);
            printf("%5d=%5d\n", code1, code2);
            exit(1);
        }
        code[i]=code1;
    }

    fclose(fpi);

    printf("the end of reading input file \n\n");

    /*Convert the data to binary*/

    for(i=0; i<=255; i++)
    {
        pwr=256;
        remainder=code[i];

        for(weight=7; weight>=0; weight--)
        {
            pwr=pwr/2;
            bin[i][weight]=remainder/pwr;
            remainder=remainder-bin[i][weight]*pwr;
        }
    }

    printf("the end of computing \n\n");

    /*Print out the results*/
    fpo=fopen("revrombin.txt", "w");
for(number=0; number<=255; number++)
{
    for(weight=7; weight>=0; weight--)
        fprintf(fpo, "%ld", bin[number][weight]);

        fprintf(fpo, "\n");
}

fclose(fpo);

printf("the end of output\n\n");
}
Appendix D

Circuitry

D.1 Introduction

This Appendix presents the work of several individuals that have worked on implementation of earlier Fermat ALU designs. The basic circuitry for the ALU is the work of Dr. Wenzhe Luo, who first implemented the Fermat ALU in a 1.5μm process. Subsequently this ALU was redesigned, using the same circuitry by Dr. Binqiao Li in 0.5μm and 0.35μm. The fabricated designs were tested and speed and power measures for the various components of the ALU were produced. Finally an optimized ROM layout was designed by Roberto Muscedere, which is suggested to be used for the modified Fermat ALU.

D.2 Adder Design

EMODL (Enhanced Multiple-Output Domino Logic) was first introduced in [109] with application to the design of efficient, fast, modular elements for carry-lookahead adders. The pseudo complement adder tree, proposed in [109], is built to a maximum height of 4 in this design. We are able to cascade these trees to obtain small bit-length (<16) adders with low power dissipation, high speed, and design modularity. The very fast but irregular
architecture reported in [109] (2.7ns critical path for a 32-bit adder) is replaced by a more modular, lower power construction that attains a sufficiently fast evaluation time to enable a 3-stage pipeline to be efficiently utilized for the complete Fermat ALU. The structure of a single level in the adder tree is shown in Figure D.1.

**Figure D.1 A single-bit level in the EMODL tree.**

In Figure D.1, $a_i$, $b_i$ are the inputs of the bit position, $S_i$ is the sum of the output; $c_i$ and $c_o$ constitute the carry chain. $\bar{a}_i$ is a normal complement and $\bar{c}_i$ is a pseudo-complement [109]. The use of pseudo-complements allows symmetrical domino circuits to be built with minimum height [109]. The XOR/XNOR gate at the top of the schematic of Figure D.1 is based on a new design proposed in [110].

The use of EMODL eliminates the p transistors associated with separate domino stages to provide an efficient logic subfunction evaluation. The modularity associated with growing the adder bit-length is shown in Figure D.2 for a 4-bit adder module; note the multiple
output sums in the single tree and the pseudo-complement carries generated by the cascaded subfunctions.

**Figure D.2 A 4-bit EMODL adder tree**

For multiple sections of 4-bits (e.g. the GF(257) MAC uses 8-bit adders) a connecting component (X-connector) is inserted between the EMODL trees to accelerate the discharge during the evaluation phase. The X-connector is functionally similar to carry regenerating buffers [66] which were used in static Manchester adders: in our case, where dual carry chains are used and the circuits are dynamic, two chains of carries need to be restored, and the ground clock switch and domino inverters are used to generate the carry signals dynamically. The X-connectors decrease the loading capacitances for the propagating carries (and their pseudo-complements), and so greatly reduce the worst-case delay. Simulation results of an 8-bit adder (made up of the two 4-bit adder trees) show a reduction in delay from 4.9ns to 3.3ns when the X-connector is inserted. An X-connector is shown in Figure D.3, with a 4n tree cascade shown in Figure D.4.
D.3 ROM Design

In the Fermat ALU, the ROM is a main component of area consumption and power dissipation, as well as the critical component limiting the throughput rate. A design of a small dynamic ROM with a power dissipation very much lower than more traditional static designs is presented here. In general, small ROMs have a disadvantage over larger ROMs in that the power dissipation of the output circuitry is only dependent on the number of output bits: large ROMs are able to amortize this area and power over the much larger number of storage bits. Our preferred design replaces the usual static circuitry with a dynamic sense amplifier and decoder.

Dynamic Sense Amplifier

The dynamic sense amplifier is shown in Figure D.5.
Figure D.5 Dynamic sense amplifier

X is a word line from the decoder. M₀ is the selected ROM cell, and M₁₋₃₁ are the unselected ROM cells of the same bit-line. MN₅ and MN₆ are one of the four column decoders, and connected to the true address bits A₁ and A₀ in this column. The presence of M₀ allows the precharged nodes to discharge producing a 1 at the output; the absence of M₀ produces a constant logic 0 at the output. The ROM programming is therefore carried out by selectively placing transistors in the storage transistor array. MP₃ is a weak p-channel pull-up transistor that reduces the charge sharing effect on the evaluation node. For this design we find that the power dissipation is very low and the area is smaller than similar ROM designs [67]. Our design reduces the power associated with a traditional differential output stage which need bias voltage and static current drain, where the static power dissipation is usually greater than the dynamic (working) power. So, in our design, the elimination of static power is a big saving for total power dissipation.

Dynamic Decoder
The Domino style decoder unit is shown in Figure D.6. We have elected to build 32 separate stages rather than the usual tree decoder. The number of transistors is greater than the tree method, but the practical layout area is the same because of the non-rectangular decoder tree structure and the layout is made more modular; the power dissipation does not increase.

**Figure D.6 Dynamic decoder unit**

![Dynamic decoder unit diagram](image)

**Minimized ROM Layout**

A typical ROM layout is shown in Figure D.7 for a target 0.35μ CMOS process. The size for a $2^7 \times 8$ ROM is $5390 \mu m^2$. 
Figure D.7 ROM layout

Table D.1 ROM size comparisons

<table>
<thead>
<tr>
<th>ROM Size</th>
<th>Area (μm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^6$ x 8</td>
<td>4600μm²</td>
</tr>
<tr>
<td>$2^7$ x 8</td>
<td>5390μm²</td>
</tr>
<tr>
<td>$2^8$ x 8</td>
<td>9652μm²</td>
</tr>
</tbody>
</table>

The area of the previously designed $2^8$ x 8 ROM (Dr. Li) is 22320μm², which is more than twice the area of the compacted ROM design for the same size.

D.4 Latch Design

The True Single Phase Clock strategy for dynamic logic [1] is used throughout the whole design. TSPC is known to suffer from edge slew problems, and we have explored this issue quite fully in order to produce robust designs.

Pipeline Latch Design

In the TSPC strategy, the clock related problems are mainly self-skewing problems, and the design of a robust latch has to involve detailed simulations with chains or counters. The TSPC latch was carefully studied in [1][117] and the clock slope dependent problems
explored in [59]. In [59], the problems related with clock slope and self skew are analyzed and sizing of transistors was suggested as a suitable way to deal with the inherent problems. In particular, an example of sizing N and P blocks concludes that the size of the precharge transistor is an important parameter in preventing edge and skew failures.

Figure D.8 is the familiar form of TSPC latch with the addition of an inverter buffer; this buffer proves indispensable to isolate the self loads of the internal sized transistors from any output circuitry. A qualitative analysis of the non-ideal clock problems with the latch, leads to the conclusion that M2-3 should be very weak to prevent false paths through M4-5 for a slow clock edge. and also to prevent problems with the input data changing during a slow clock transition. M6 should also be weak to prevent false discharge through M7-8 [59], and also M7-8 should be weak to provide sufficient hold time for a following TSPC latch stage.

![Figure D.8 A Buffered TSPC Latch](image)

Pipeline design template

For an all TSPC design, we consider the robust implementation of pipelined logic in dynamic circuits. In [58], the subject of skew and logic flexibility are explored, a set of rules for dynamic block connections are suggested, and various timing strategies provided. From this work we have produced a pipeline design template, as shown in Figure D.9, which is suitable for the long pipeline cascades projected as implementation architectures.
for the Fermat ALU. In Figure D.9 all of the logic functions are implemented in the N switching trees [46] of the preceding domino stages. The output signals of the latches may also be inverted to provide extra logic flexibility in the domino switching trees [46]. The Fermat ALU is designed as a three pipeline stage in this fashion, with the adders and ROM components formed as the n-channel logic blocks in Figure D.9. The transistors in the latch were sized to overcome the self-skewing problem of TSPC designs [48].

Figure D.9 Domino stages with TSPC latch

D.5 Fermat ALU Layout

Figure D.10 shows an initial layout of the Fermat ALU in a 0.35μm triple metal process. This test chip was designed to evaluate the properties of the architecture rather than to minimize layout area.
Table D.2 shows the power estimation for the various components of the Fermat ALU in both a 0.5μm CMOS and the target 0.35μm process.

<table>
<thead>
<tr>
<th></th>
<th>Power (mW/100MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.5μm</td>
</tr>
<tr>
<td>8-bits adder</td>
<td>1.24</td>
</tr>
<tr>
<td>256x8 bits ROM</td>
<td>1.44</td>
</tr>
<tr>
<td>128x8 bits ROM</td>
<td>0.89</td>
</tr>
<tr>
<td>DFF</td>
<td>0.05</td>
</tr>
<tr>
<td>Original Fermat275</td>
<td>5.12</td>
</tr>
<tr>
<td>New Fermat 257</td>
<td>4.57</td>
</tr>
<tr>
<td>MAC with Booth algorithm</td>
<td>41.0</td>
</tr>
</tbody>
</table>

All components of the processor have been designed with minimum power dissipation at video processing rates as a goal. The output delay of the 8-bit dynamic adder is 3.65ns and 3.25ns for the 0.5μm and 0.35μm designs respectively. The pipeline processor structure can operate at maximum clock frequencies of 120MHz and 133 MHz for the 0.5μm and
0.35μm designs respectively. The outputs of a low cycle rate functional test and delay test are shown in Figure D.11.

**Figure D.11 Low cycle rate test and Delay test**

---

D.6 Comparison with a Binary MAC

In order to compare the power dissipation of the Fermat ALU with an equivalent binary MAC, we have produced an accurate simulation of a high performance binary MAC with an equivalent dynamic range. We have assumed the use of a 24-bit accumulator, with scaling employed for large numbers of coefficients. The comparison MAC contains a 10 × 10 multiplier along with a 24-bit accumulator. The power budget for the binary MAC is shown in Table D.3

**Table D.3 Power Budget for the Binary MAC**

<table>
<thead>
<tr>
<th>Component</th>
<th>Power(mW/100MHz)</th>
<th>Power(mW/100MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.5μm</td>
<td>0.35μm</td>
</tr>
<tr>
<td>Full Adder Array</td>
<td>18.35</td>
<td>15.2</td>
</tr>
<tr>
<td>Carry Lookahead Adder</td>
<td>8.25</td>
<td>7.01</td>
</tr>
<tr>
<td>Booth selector</td>
<td>5.95</td>
<td>4.76</td>
</tr>
<tr>
<td>Booth encoder</td>
<td>3.65</td>
<td>2.92</td>
</tr>
<tr>
<td>Carry Chain</td>
<td>3.3</td>
<td>2.4</td>
</tr>
<tr>
<td>Input Driver</td>
<td>2.3</td>
<td>1.71</td>
</tr>
<tr>
<td>Total</td>
<td>41.0</td>
<td>34.0</td>
</tr>
</tbody>
</table>
explored in [59]. In [59], the problems related with clock slope and self skew are analyzed and sizing of transistors was suggested as a suitable way to deal with the inherent problems. In particular, an example of sizing N and P blocks concludes that the size of the precharge transistor is an important parameter in preventing edge and skew failures.

Figure D.8 is the familiar form of TSPC latch with the addition of an inverter buffer; this buffer proves indispensable to isolate the self loads of the internal sized transistors from any output circuitry. A qualitative analysis of the non-ideal clock problems with the latch, leads to the conclusion that M2-3 should be very weak to prevent false paths through M4-5 for a slow clock edge, and also to prevent problems with the input data changing during a slow clock transition. M6 should also be weak to prevent false discharge through M7-8 [59], and also M7-8 should be weak to provide sufficient hold time for a following TSPC latch stage.

**Figure D.8 A Buffered TSPC Latch**

Pipeline design template

For an all TSPC design, we consider the robust implementation of pipelined logic in dynamic circuits. In [58], the subject of skew and logic flexibility are explored, a set of rules for dynamic block connections are suggested, and various timing strategies provided. From this work we have produced a pipeline design template, as shown in Figure D.9, which is suitable for the long pipeline cascades projected as implementation architectures.
for the Fermat ALU. In Figure D.9 all of the logic functions are implemented in the N switching trees [46] of the preceding domino stages. The output signals of the latches may also be inverted to provide extra logic flexibility in the domino switching trees [46]. The Fermat ALU is designed as a three pipeline stage in this fashion, with the adders and ROM components formed as the n-channel logic blocks in Figure D.9. The transistors in the latch were sized to overcome the self-skewing problem of TSPC designs [48].

Figure D.9 Domino stages with TSPC latch

D.5 Fermat ALU Layout

Figure D.10 shows an initial layout of the Fermat ALU in a 0.35µm. triple metal process. This test chip was designed to evaluate the properties of the architecture rather than to minimize layout area.
Table D.2 shows the power estimation for the various components of the Fermat ALU in both a 0.5μm CMOS and the target 0.35μm process.

Table D.2 Power Estimation

<table>
<thead>
<tr>
<th>Component</th>
<th>Power (mW/100MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.5μm</td>
</tr>
<tr>
<td>8-bits adder</td>
<td>1.24</td>
</tr>
<tr>
<td>256x8 bits ROM</td>
<td>1.44</td>
</tr>
<tr>
<td>128x8 bits ROM</td>
<td>0.89</td>
</tr>
<tr>
<td>DFF</td>
<td>0.05</td>
</tr>
<tr>
<td>Original Fermat275</td>
<td>5.12</td>
</tr>
<tr>
<td>New Fermat 257</td>
<td>4.57</td>
</tr>
<tr>
<td>MAC with Booth algorithm</td>
<td>41.0</td>
</tr>
</tbody>
</table>

All components of the processor have been designed with minimum power dissipation at video processing rates as a goal. The output delay of the 8-bit dynamic adder is 3.65ns and 3.25ns for the 0.5μm and 0.35μm designs respectively. The pipeline processor structure can operate at maximum clock frequencies of 120MHz and 133 MHz for the 0.5μm and
0.35μm designs respectively. The outputs of a low cycle rate functional test and delay test are shown in Figure D.11.

**Figure D.11 Low cycle rate test and Delay test**

![Diagram showing waveforms for low cycle rate test and delay test.]

### D.6 Comparison with a Binary MAC

In order to compare the power dissipation of the Fermat ALU with an equivalent binary MAC, we have produced an accurate simulation of a high performance binary MAC, with an equivalent dynamic range. We have assumed the use of a 24-bit accumulator, with scaling employed for large numbers of coefficients. The comparison MAC contains a 10 × 10 multiplier along with a 24-bit accumulator. The power budget for the binary MAC is shown in Table D.3

**Table D.3 Power Budget for the Binary MAC**

<table>
<thead>
<tr>
<th>Component</th>
<th>Power(mW/100MHz) 0.5μm</th>
<th>Power(mW/100MHz) 0.35μm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Adder Array</td>
<td>18.35</td>
<td>15.2</td>
</tr>
<tr>
<td>Carry Lookahead Adder</td>
<td>8.25</td>
<td>7.01</td>
</tr>
<tr>
<td>Booth selector</td>
<td>5.95</td>
<td>4.76</td>
</tr>
<tr>
<td>Booth encoder</td>
<td>3.65</td>
<td>2.92</td>
</tr>
<tr>
<td>Carry Chain</td>
<td>3.3</td>
<td>2.4</td>
</tr>
<tr>
<td>Input Driver</td>
<td>2.3</td>
<td>1.71</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>41.0</strong></td>
<td><strong>34.0</strong></td>
</tr>
</tbody>
</table>
Vita Auctoris

Marjan Shahkarami, born 17 December, 1967 in Abadan, Iran. She received the Bachelor of Applied Science in Telecommunication Engineering from K.N. Toosi University, Tehran, Iran in 1990 and the MaSc. in Electrical Engineering from the University of Windsor, Windsor, ON, Canada, in 1994. She is a candidate in the electrical engineering Ph.D program at the University of Windsor. Her research areas include high performance VLSI circuit design, computer arithmetic, fault tolerant design and digital signal processing.