Lossy Polynomial
Datapath Synthesis

Theo Drane

February 2014

Supervised by George A. Constantinides
and Peter Y. K. Cheung

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Electrical and Electronic Engineering of Imperial
College London
and the Diploma of Imperial College London
The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.
Abstract

The design of the compute elements of hardware, its datapath, plays a crucial role in determining the speed, area and power consumption of a device. The building blocks of datapath are polynomial in nature. Research into the implementation of adders and multipliers has a long history and developments in this area will continue. Despite such efficient building block implementations, correctly determining the necessary precision of each building block within a design is a challenge. It is typical that standard or uniform precisions are chosen, such as the IEEE floating point precisions. The hardware quality of the datapath is inextricably linked to the precisions of which it is composed. There is, however, another essential element that determines hardware quality, namely that of the accuracy of the components. If one were to implement each of the official IEEE rounding modes, significant differences in hardware quality would be found. But in the same fashion that standard precisions may be unnecessarily chosen, it is typical that components may be constructed to return one of these correctly rounded results, where in fact such accuracy is far from necessary. Unfortunately if a lesser accuracy is permissible then the techniques that exist to reduce hardware implementation cost by exploiting such freedom invariably produce an error with extremely difficult to determine properties.

This thesis addresses the problem of how to construct hardware to efficiently implement fixed and floating-point polynomials while exploiting a global error freedom. This is a form of lossy synthesis. The fixed-point contributions include resource minimisation when implementing mutually exclusive polynomials, the construction of minimal lossy components with guaranteed worst case error and a technique for efficient composition of such components. Contributions are also made to how a floating-point polynomial can be implemented with guaranteed relative error.
Acknowledgements

This thesis is another strand that links Imagination Technologies with Imperial College London’s Department of Electrical and Electronic Engineering. Its inception is a testament to the vision of Imagination Technologies’ Tony King-Smith and Martin Ashton and Imperial College London’s Peter Cheung and George Constantinides and for its execution it owes much to many.

The thesis has grown at the same time that Imagination’s Datapath group has grown. I have been incredibly fortunate to find six incredibly talented engineers who, luckily for me, took an interest in a field which I fell into quite by accident many years ago and have since made my home. For all their work, enthusiasm, ingenuity, tenacity and particular independence I sincerely thank Casper Van Benthem, Freddie Exall, Thomas Rose, Sam Elliott, Kenneth Rovers and Emiliano Morini. Particular mention goes to Freddie’s staggering array of essential tools TBGen, Hectorator, Paretor and Decorator, Thomas for FRator, Casper and Kenneth’s implementations of faithfully rounded floating-point multipliers and adders respectively and Sam’s floating-point verification assistance. All without which this work would not have been possible. Thanks also to Imagination’s Raeeke Yassaie and Imperial’s Clare Drysdale for all of their support and fascinating conversations on all matters industrial academic. Gratitude goes to Clifford Gibson and Jonathan Redshaw for being very understanding line managers as I tried to navigate two complementary yet competing sets of obligations.

I would like to thank Carl Pixley and Himanshu Jain whose development of the paradigm shifting Synopsys’ Hector has enabled my group to flourish, provided fascinating insight into our verification challenges and enabled me to share a verification success story in multiple countries.

On the Imperial side, sincere thanks goes to David Boland, Samuel Bayliss and Josh Levine of the Circuits and Systems Group for their technical insight, critique and refreshingly broad conversation. Being involved in the shadow management of the EDA club beside David and Sam was a very
interesting journey, thanks to my companions for the ride. Thanks to EDA members for enriching my knowledge by the various talks and David Thomas for sharing his interesting work, unique approach to the field, clarity of presentation and critique. I offer my supervisor, George Constantinides, my sincere gratitude for pushing this work to be the best it can be, his insight, vision, direction and incredible ability to quickly understand ideas, despite being presented, on occasion, with truly incomprehensible material.

It is essential to thank my parents for giving me blank paper from a young age and luckily we all still use it. Finally it remains to thank my partner, James Van Der Walt, for his unending support and whose own work constantly reminded me that ultimately we do our work to improve the lives of others. I believe my work has enriched the field, improved and pushed my group’s work, added value to the company and its products. This thesis has its roots in the continuous design challenges thrown up on a daily basis by working in the mobile graphics hardware sector. It is a truly exciting area for computer arithmetic, large amounts of datapath constrained by ever changing constraints on power, area, speed, precision and accuracy. Hopefully this thesis contributes a little to giving engineers the tools they need to do what must be done. I hope aspects of this thesis will live on in tools and code within Imagination, will be extended and improved by others and ultimately find its way into devices used by many. To those who may be interested in having one foot firmly in an academic setting and another in an industrial one and who wishes to stand and nudge the tide in both, I would say that it is one of the best things I have ever done.
Contents

1. Introduction ........................................... 22
   1.1. Statement of Originality ............................. 25
   1.2. Impact ............................................ 25
   1.3. Thesis Organisation ................................. 28

2. Background ............................................ 30
   2.1. Logic and Datapath Synthesis ...................... 32
      2.1.1. Integer Addition and Multiplication ........ 35
      2.1.2. Datapath Synthesis Flow ....................... 43
   2.2. High-Level Synthesis ............................... 44
   2.3. Lossy Synthesis ................................... 49
      2.3.1. Word-length Optimisation ....................... 49
      2.3.2. Imprecise Operators ............................ 51
      2.3.3. Gate Level Imprecision ......................... 53
   2.4. Polynomials in Datapath ............................ 55
   2.5. Context of Contribution ............................. 57

3. Preliminaries .......................................... 60
   3.1. Polynomials and Polynomial Rings ................. 61
   3.2. Multivariate Division .............................. 63
   3.3. Gröbner Bases .................................... 67
   3.4. Elimination and Solutions to Systems of Multivariate Polynomials ........................................... 70
   3.5. Membership of Vanishing Ideals .................... 73

4. Lossless Fixed-Point Polynomial Optimisation ............... 79
   4.1. Arithmetic Sum-Of-Products ......................... 82
   4.2. Motivational Example ............................... 85
4.3. Control Logic Minimisation .................................. 88
  4.3.1. Determining the Total Degrees of Monomials in $p$ . . . 89
  4.3.2. Solving the Optimisation Problem .......................... 92
  4.3.3. Lifting the Restrictions .................................. 95
  4.3.4. Changing Variables ...................................... 98
4.4. Optimising the Optional Negations ........................... 100
4.5. Overall Flow .................................................. 105
4.6. Formal Verification ............................................ 107
  4.6.1. The Waterfall Verification ................................ 112
  4.6.2. Overall Verification Flow ................................. 113
4.7. Experiments .................................................... 116
  4.7.1. Control Logic Minimisation Experiments .................. 117
  4.7.2. Optional Negation Experiments ........................... 124
  4.7.3. General Experiments .................................... 128
  4.7.4. Formal Verification Experiments ........................ 135

5. Lossy Fixed-Point Components .................................. 137
  5.1. Integer Multiplication ...................................... 139
    5.1.1. Background on Truncation Multiplier Schemes .......... 139
    5.1.2. Constructing Faithfully Rounded Multipliers .......... 143
    5.1.3. Error Properties of Truncated Multipliers ............ 154
  5.2. Multioperand Addition ...................................... 157
    5.2.1. Optimal Truncation Theorem ............................ 159
    5.2.2. Lemma 1: $l_{i}^{opt} = h_i$ for $i < k$ ............... 160
    5.2.3. Lemma 2: $l_{i}^{opt} = 0$ for $i > k$ .................. 161
    5.2.4. Faithfully Rounded Array Theorem ..................... 162
    5.2.5. Application to Multiplier Architectures ............... 163
    5.2.6. Experiments .......................................... 166
  5.3. Constant Division .......................................... 172
    5.3.1. Conditions for Round Towards Zero Schemes .......... 173
    5.3.2. Conditions for Round To Nearest, Even and Faithfully
           Rounded Schemes ...................................... 175
    5.3.3. Hardware Cost Heuristic ................................ 176
    5.3.4. Optimal Round Towards Zero Scheme ................... 176
    5.3.5. Optimal Rounding Schemes ............................. 179
    5.3.6. Extension to Division of Two’s Complement Numbers 179
5.3.7. Application to the Multiplication of Unsigned Normalised Numbers .................................. 180
5.3.8. Experiments ............................................................................................................. 181
5.4. Formal Verification of Lossy Components ..................................................................... 182

6. Lossy Fixed-Point Polynomial Synthesis ........................................................................... 185
   6.1. Exploiting Errors for Arbitrary Arrays and Constant Division ................................. 187
   6.2. Array Constructions .................................................................................................. 189
   6.3. Navigation of the Error Landscape ............................................................................. 192
       6.3.1. Developing a Heuristic for the Navigation of the Error Landscape .................. 193
       6.3.2. Example Navigations of the Error Landscape .................................................. 196
       6.3.3. Procedure for Error Landscape Navigation ....................................................... 210
   6.4. Fixed-Point Polynomial Hardware Heuristic Cost Function ..................................... 214
       6.4.1. Binary Integer Adder .......................................................................................... 214
       6.4.2. Binary Rectangular Arrays .............................................................................. 215
       6.4.3. Arbitrary Binary Array ..................................................................................... 217
       6.4.4. Multipliers ......................................................................................................... 217
       6.4.5. Squarers ............................................................................................................ 218
       6.4.6. Constant Multiplication .................................................................................... 219
       6.4.7. Example Heuristic Calculation Cost ................................................................... 219
   6.5. Experiments ............................................................................................................... 223
       6.5.1. Comparison of Word-length Optimisation versus Array Truncation ................... 224
       6.5.2. Validation of Area Heuristic Cost Function ....................................................... 232
       6.5.3. Quality of Error Landscape Navigation ............................................................. 233
       6.5.4. Conclusion .......................................................................................................... 238

7. Lossy Floating-Point Polynomial Synthesis ....................................................................... 240
   7.1. Floating-Point Background and Algorithms for Implementing Polynomials ................. 241
       7.1.1. Error-Free Transformations ............................................................................... 243
       7.1.2. Polynomial Evaluation ...................................................................................... 244
   7.2. Lossy Floating-Point Components .............................................................................. 249
       7.2.1. Lossy Floating-Point Multipliers ....................................................................... 249
       7.2.2. Faithfully Rounded Floating-Point Multiplier Theorem ...................................... 250
B. Faithfully Rounded FP Addition

C. The Condition for the Use of F32/F64 Far from \( x = y = z \) 330
   C.1. The Conditions for the Use of F32 . . . . . . . . . . . . . . . 330
   C.2. The Conditions for the Use of F64 . . . . . . . . . . . . . . . 333

D. The Condition for the Use of F32/F64 Far from \( z = x = 0 \) 337
   D.1. The Conditions for the Use of F32 . . . . . . . . . . . . . . . 337
   D.2. The Conditions for the Use of F64 . . . . . . . . . . . . . . . 342

E. The Condition for the Use of F32/F64 Far from the Variety 347
   E.1. The Conditions for the Use of F32 . . . . . . . . . . . . . . . 347
   E.2. The Conditions for the Use of F64 . . . . . . . . . . . . . . . 354

F. Calculation of \( \Delta \) and \( p_{\min} \) Near \( x = y = z \) 362
   F.1. Calculation of \( \Delta \) . . . . . . . . . . . . . . . . . . . . . . 362
   F.2. Calculation of \( p_{\min} \) . . . . . . . . . . . . . . . . . . . . . 365

G. Calculation of \( \Delta \) and \( p_{\min} \) Near \( x = z = 0 \) 368
   G.1. Calculation of \( \Delta \) . . . . . . . . . . . . . . . . . . . . . . 368
   G.2. Calculation of \( p_{\min} \) . . . . . . . . . . . . . . . . . . . . . 372

H. Calculation of \( p_{\min} \) Just Off the Origin and Far from the Variety 374
## List of Tables

2.1. Commercial ASIC Synthesis Tools. .......................... 33  
2.2. Commercial LEC Tools. ................................. 33  
2.3. Datapath Synthesis Tools and Libraries. ................. 34  
2.4. High-Level Synthesis Tools. ............................. 46  

4.1. Synthesis Results for Sample SOPs. ....................... 84  
4.2. Integer Program Runtimes. ............................... 123  
4.3. Integer Program Runtimes. ............................... 132  
4.4. Area Benefits of the Proposed Flow. ..................... 135  
4.5. Formal Verification Runtimes. ............................ 136  

5.1. X values for VCT worst case error vectors. .............. 154  
5.2. X values for VCT worst case error vectors. .............. 157  
5.3. $X^\pm$ and $Y^\pm$ for RTZ, RTE and FR. ................. 179  

6.1. Area cost of Parallel Prefix Adder. ...................... 215  
6.2. Characteristics of the Partial Product Arrays of a Cubic Polynomial Implementation. ......................... 220  
6.3. Cubic Polynomial Heuristic Area Cost. .................... 221  
6.4. Runtimes of Hector Lemmas Determining Non Zero Partial Product Bits. ........................................... 225  
6.5. Correlation Coefficients for Normalised Area versus Heuristic Area Cost Function. ............................. 233  

7.1. Floating-Point Types. ...................................... 242  
7.2. Floating-Point Interpretation. ............................. 242  
7.3. Accurate Evaluability of Polynomials. .................... 247  
7.4. Allowable Varieties and Accurate Evaluability of Polynomials. ........................................ 256  
7.5. Values of $e$ for Lines of the Variety and F32 and F64. .. 267  
7.6. Constraints for Input Domain Partitioning. ............... 282
List of Figures

2.1. Intel CPU Introductions [161]. ................................. 31
2.2. Levels of Design Abstraction. ................................. 32
2.3. Average Number of gates of Logic and Datapath, Excluding
    memories [59]. ............................................. 34
2.4. Equivalence Checking with Hint files. ....................... 35
2.5. Structure of AND Multiplication Array. ..................... 39
2.6. Initial Structure of Booth Radix-4 Multiplication Array. 41
2.7. Simplified Structure of Booth Radix-4 Multiplication Array. 41
2.8. Structure of Booth Radix-4 Multiplication Array. 41
2.9. Array Reduction of 5 bit AND Array Multiplier. .......... 42
2.10. Lossy Synthesiser. ......................................... 58

3.1. Multivariate Division Algorithm. .............................. 66
3.2. Buchberger’s Algorithm. ...................................... 68

4.1. Example Sum-Of-Products Array. ............................... 82
4.2. Example Sum-Of-Products Array Reduction. .................. 83
4.3. Exampel Data-Flow Graph. ................................... 110
4.4. Polynomial Behaviour of Operators (P=Polynomial and N=Non-
    Polynomial). .............................................. 110
4.5. Algorithm for Establishing Polynomial Nature of Inputs. .. 111
4.6. Augmented Data-Flow Graph. ................................ 111
4.7. Formal Verifications Required. .............................. 113
4.8. Area-Delay curves for $m_1$, $m_5$ and $AB + C$ which forms a
    lower bound on achievable area. .......................... 117
4.9. Area-Delay curves for $y_{1,1}$ and $y_{1,2}$. .................. 122
4.10. Area-Delay curves for $y_{2,1}$ and $y_{2,2}$. ................ 122
4.11. Area-Delay curves for $y_{3,1}$ and $y_{3,2}$. ................ 123
4.12. Area-Delay curves for $y_1$ pre and post the optimisation. . 125
4.13. Area-Delay curves for $y_2$ pre and post the optimisation. . . . 125
4.14. Area-Delay curves for $y_3$ pre and post the optimisation. . . . 126
4.15. Area-Delay curves for $y_4$ pre and post the optimisation. . . . 126
4.16. Area-Delay curves for $y_5$ pre and post the optimisation. . . . 128
4.17. Area-Delay curves for $y_{1,0}$ and $y_{1,2}$. . . . . . . . . . . . . . . . . 129
4.18. Area-Delay curves for $y_{2,0}$ and $y_{2,2}$. . . . . . . . . . . . . . . . . . 130
4.19. Area-Delay curves for $y_{3,0}$ and $y_{3,2}$. . . . . . . . . . . . . . . . . . 130
4.20. Area-Delay curves for $y_{6,0}$ and $y_{6,1}$. . . . . . . . . . . . . . . . . . 133
4.21. Area-Delay curves for $y_{7,0}$ and $y_{7,1}$. . . . . . . . . . . . . . . . . . 134

5.1. Structure of AND Array Multiplier Truncation Schemes. . . . 139
5.2. Structure of an arbitrary array truncation scheme. . . . . . . 158
5.3. Illustration of $h_i$ of $l_i$. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 159
5.4. Optimal Truncation of an Arbitrary Array. . . . . . . . . . . . 163
5.5. Ragged Truncated Multiplier — AND Array. . . . . . . . . . . 164
5.6. Booth Radix-4 Array — Least Significant Columns. . . . . . 164
5.7. Ragged Truncated Multipliers — Booth Radix-4 Array. . . . . 165
5.8. Number of Saved Partial Product Bits over CCT. . . . . . . . . 167
5.9. Area/Delay Comparisons of Faithfully Rounded AND Array
    Multipliers n=16. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 168
5.10. Area/Delay Comparisons of Faithfully Rounded AND Array
    Multipliers n=24. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 168
5.11. Area/Delay Comparisons of Faithfully Rounded AND Array
    Multipliers n=32. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 169
5.12. Area/Delay Comparisons of Faithfully Rounded Booth Array
    Multipliers n=16. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 169
5.13. Area/Delay Comparisons of Faithfully Rounded Booth Array
    Multipliers n=24. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 170
5.14. Area/Delay Comparisons of Faithfully Rounded Booth Array
    Multipliers n=32. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 170
5.15. Example Linearly Bounded Sawtooth Function. . . . . . . . . . 174
5.16. Delay and Area Synthesis Comparisons. . . . . . . . . . . . . . . 182
5.17. Verification Runtimes for Faithfully Rounded CCT Multipliers
    of Size $n$. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 183
5.18. Verification Runtimes for Faithfully Rounded Division by
    Constant $n$. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 184
6.1. Example SOP and Constant Division DFG. ................................. 192
6.2. Basic Implementation of Cubic Polynomial. .............................. 197
6.3. 16 bit Truncated Squarer Array. ........................................... 199
6.4. 16 by 18 bit Truncated Multiplier Array. ............................... 199
6.5. Truncated SOP Array. ....................................................... 200
6.6. DFG of Bilinear Interpolation. ............................................ 202
6.7. DFG of Gabor Filter. ....................................................... 207
6.8. Reduction of a Binary Rectangular Array. ................................ 216
6.9. 10 by 10 Multiplier Booth Radix-4 Array. ............................... 217
6.10. Flow for Truncating an HDL Array. .................................... 224
6.11. Cubic Polynomial Area-Delay Curves. .................................... 226
6.12. Cubic Polynomial Area-Delay Curves — Starting State 2. ......... 227
6.15. Bilinear Interpolation Area-Delay Curves — Starting State 2. .... 229
6.16. Bilinear Interpolation Area-Delay Curves — Starting State 3. .... 229
6.17. Gabor Filter Area-Delay Curves. ......................................... 230
6.18. Gabor Filter Area-Delay Curves — Starting State 2. ............... 231
6.20. Sample DFG Highlighting a Limitation of the Area Heuristic
     Cost Function. .............................................................. 233
6.21. Area-Delay Curves for the Start and End Implementations of
     the Iterative Procedure for the Cubic Polynomial. .................... 234
6.22. Area-Delay Curves for the Start and End Implementations of
     the Iterative Procedure for the Bilinear Interpolation. .............. 235
6.23. Area-Delay Curves for the Start and End Implementations of
     the Iterative Procedure for the Gabor Filter. .......................... 235
6.24. Zoom of Area-Delay Curves for the Start and End Implementations
     of the Iterative Procedure for the Gabor Filter. ...................... 236
6.25. Area-Delay Curves for the Arbitrary, Start and End Implementations
     for the Cubic Polynomial. ................................................ 237
6.26. Area-Delay Curves for the Arbitrary, Start and End Implementations
     for the Bilinear Interpolation. .......................................... 237
6.27. Area-Delay Curves for the Arbitrary, Start and End Implementations
     for the Gabor Filter. .................................................... 238
7.1. Area-Delay Curves for F16 Multiplier Architectures. . . . . . 253
7.2. Area Delay Curves for F32 Multiplier Architectures. . . . . . 253
7.3. The Variety of the Motzkin Polynomial. . . . . . . . . . . 258
7.4. Projection of the Variety of the Motzkin Polynomial onto the 
   \( xy \) Plane. . . . . . . . . . . . . . . . . . . . . . . . . . . 259
7.5. Projection of the Variety of the Motzkin Polynomial onto the 
   \( yz \) Plane. . . . . . . . . . . . . . . . . . . . . . . . . . . 260
7.6. Projection of the Variety of the Motzkin Polynomial onto the 
   \( xz \) Plane. . . . . . . . . . . . . . . . . . . . . . . . . . . 261
7.7. Unit Sphere and Curved Pyramid around the line \( x = y = z \). 266
7.8. Points \((i, j)\) if \(a^i b^j\) exists in \(p(z, a, b)\). . . . . . . . . 269

B.1. Area Delay Curves for F16 Adder Architectures. . . . . . 328
B.2. Area Delay Curves for F32 Adder Architectures. . . . . . 329
# Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>AT</td>
<td>arithmetic transform, polynomial with integer coefficients and binary inputs representing a circuit</td>
</tr>
<tr>
<td>BDD</td>
<td>Binary Decision Diagram</td>
</tr>
<tr>
<td>DFG</td>
<td>Data Flow Graph</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain Specific Language</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>EDA</td>
<td>Electronic Design Automation</td>
</tr>
<tr>
<td>Elimination Order</td>
<td>term ordering placing a set of variables higher than another</td>
</tr>
<tr>
<td>Full Adder</td>
<td>Logic cell which produces the sum of three bits</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General Purpose Graphics Processing Unit</td>
</tr>
<tr>
<td>Half Adder</td>
<td>Logic cell which produces the sum of two bits</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>HLS</td>
<td>High-Level Synthesis</td>
</tr>
<tr>
<td>Homogeneous Polynomial</td>
<td>a polynomial whose monomials all have the same total degree</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit</td>
</tr>
<tr>
<td>Ideal</td>
<td>the span of a set of polynomials</td>
</tr>
<tr>
<td>Ideal Membership Testing</td>
<td>testing whether a given polynomial belongs to an ideal</td>
</tr>
<tr>
<td>LEC</td>
<td>Logic Equivalence Checking</td>
</tr>
<tr>
<td>lex</td>
<td>Lexicographic Term ordering</td>
</tr>
<tr>
<td>Monomial</td>
<td>a product of variables, a term in a polynomial</td>
</tr>
<tr>
<td>Partial Product</td>
<td>a row in a multiplier of sum-of-products array</td>
</tr>
<tr>
<td>QoR</td>
<td>Quality of Results</td>
</tr>
<tr>
<td>ROBDD</td>
<td>Reduced Order Binary Decision Diagram</td>
</tr>
<tr>
<td>RTE</td>
<td>Round Towards Nearest, Ties to Even</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>RTZ</td>
<td>Round Towards Zero</td>
</tr>
<tr>
<td>Term</td>
<td>Definition/Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SEC</td>
<td>Sequential Equivalence Checking</td>
</tr>
<tr>
<td>Smarandache Function</td>
<td>number theoretic function, for a given positive integer n it returns the smallest number which divides its factorial</td>
</tr>
<tr>
<td>SMT</td>
<td>Satisfiability Modulo Theories</td>
</tr>
<tr>
<td>SOP</td>
<td>Arithmetic Sum-of-Products</td>
</tr>
<tr>
<td>Term Order</td>
<td>ordering applied to monomials</td>
</tr>
<tr>
<td>Total degree</td>
<td>the sum of exponents in a monomial</td>
</tr>
<tr>
<td>Vanishing Ideal</td>
<td>ideal of polynomials which are identically zero over the ring of polynomials</td>
</tr>
<tr>
<td>Variety</td>
<td>the set of points, such that for all polynomials in an ideal, they evaluate to zero on those points</td>
</tr>
</tbody>
</table>
Nomenclature

\[ y[n : m] \]: bits \(n\) down to \(m\) in the binary representation of variable \(y\)
\[ y[i] \text{ or } y_i \]: a bit of variable \(y\), \(y_0\) being the least significant bit
\[ \mathbb{Q} \]: set of rational numbers
\[ \mathbb{R} \]: set of real numbers
\[ \mathbb{Z} \]: set of integers
\[ \mathbb{Z}_p \]: set of integers modulo \(p\)
\[ \text{deg}(p) \]: largest exponent in univariate polynomial \(p\)
\[ k[x_1, x_2, \ldots, x_n] \]: polynomial ring in variables \(x_1, \ldots, x_n\) with coefficients in set \(k\)
\[ \langle f_1, f_2, \ldots, f_s \rangle \]: ideal - the span of a set of polynomials \(f_1, f_2, \ldots, f_s\)
\[ p|q \]: condition that \(p\) divides \(q\), if they are polynomials \(p, q \in k[x]\)

then there must exist \(r \in k[x]\) such that \(q = pr\)
\[ LP(p) \]: leading power product of \(p\) with respect to a term ordering
\[ LC(p) \]: leading coefficient of \(p\) with respect to a term ordering
\[ LT(p) \]: leading term of \(p\) with respect to a term ordering
\[ r^\text{vanish}_p \]: vanishing ideal over the ring \(\mathbb{Z}_p\)
\[ (n)_r \]: falling factorial \((n)_r = n(n-1)(n-2)\ldots(n-r+1)\)
\[ SF(p) \]: Smarandache Function, returns the smallest \(n\) such that \(p|n!\)
\[ \#_{p}(k) \]: number of times \(p\) divides \(k\)
\[ \text{Ham}(n) \]: Hamming Weight of \(n\)
\[ V(I) \]: affine variety of ideal \(I\), the set of points, \(x\), such that for all \(f \in I\), \(f(x) = 0\)
\[ s?a : b \]: ternary operator, if \(s\) then \(a\) else \(b\)
\[ |S| \]: denotes the number of elements in the set \(S\)
\[ \overline{a} \]: bitwise negation of binary variable \(a\)
\[ \lor \]: logical OR
\[ \land \]: logical AND
\[ a \oplus b \]: logical XOR, if \(a\) and \(b\) are of equal length then this represents a bitwise XOR, if \(a\) is of length greater than one and \(b\) is of length one then this represents XORing each bit of \(a\) with \(b\)
$a \ll b$ left binary shift of variable $a$ by variable $b$

$a \gg b$ right binary shift of variable $a$ by variable $b$

FA Full Adder cell which produces the sum of three bits

HA Half Adder which produces the sum of two bits

$sgn(x)$ sign of $x$, -1 if $x < 0$, 1 if $x > 0$ and 0 if $x = 0$

RTE round towards nearest, ties to even rounding mode

RTZ round towards zero rounding mode
1. Introduction

Electronic systems pervade our daily lives, from our reliance on the internet, the extraordinary growth of mobile communications to the many sophisticated ways in which we now consume information, whether written, audio or visual. In order to construct the devices which make this a reality a range of challenges have had to be overcome. Internal power management for one, where minimising energy usage means the ability to work from battery power and thus enabling portability. Many will have advanced graphics and human-machine interfaces which need to be of high enough quality and responsiveness to enable market success [170]. Portability and quality requirements translate into speed, area and power constraints on the underlying hardware. Notions of quality can include image or audio properties, high enough quality means minimum throughput capacity, memory bandwidth and accuracy requirements. Certain notions of quality are embedded in standards which new devices entering the market must satisfy, compliance with standards means passing a variety of conformance tests. Video standards include H264 [83] and MPEG4 [168], graphics standards include OpenGL [94] and frameworks for targeting heterogeneous platforms include OpenCL [93]. Devices that succeed in the market place must conform to all relevant standards and given that people, and hence devices, move between countries, these standards come from all over the world, compete with each other and continuously evolve. Hardware designers must produce multi-standard hardware with acceptable throughput, frequency, area and power properties in a short enough timescale in order for products to hit the market at an opportune time. Designers must translate all of the hardware requirements into a hardware description, explore the various trade offs that exist and perform verification at speed.

This translation exercise is surprisingly non trivial for even the simplest of designs. Consider creating hardware that calculates $\sin(x)$ that must support multiple standards, each with their own precision, accuracy and
throughput requirements, in minimal silicon area. A translation exercise requires selecting the number format, algorithm and associated architecture and the precision and accuracy of each operation. All aspects of this task are non trivial, even though the number format for the inputs and outputs may be determined by the application, internal formats need not necessarily match these. For power constrained devices, it is not enough that the precisions are sufficient for the required accuracy, there must be just enough precision. Using standard precisions, such as IEEE single or double precision floating-point formats [1], in an effort to match a software implementation, may be extremely wasteful in terms of silicon or potentially provide not enough precision. Designing application specific integrated circuits (ASICs) allows for the use of any number format, any precision and any accuracy — to ignore this freedom is to systematically waste silicon, degrade performance and potentially quality. Using such freedom is a great opportunity and a great challenge.

This thesis deals with the design of datapath elements in ASIC hardware design. Certain integrated circuits are necessarily constructed with a large amount of datapath, such as graphics processing units (GPUs). In these areas, datapath often determines the maximum operating frequency, contributes significantly to the area and power consumption as well as presenting challenges to the verification effort. Considerable research has gone into the fundamental arithmetic operations, namely addition and multiplication and leading industry logic synthesis tools are highly adept at creating very efficient implementations when the design is polynomial in nature. Examples of such synthesis tools are Synopsys’ Design Compiler [162] and Cadence’s RTL Compiler [19]. Algorithms have been developed which exploit these efficient functions. For example, it is common to use polynomials to approximate transcendental functions like \( \sin(x) \). Polynomial functions can be efficiently implemented and are often required, as such, they are the focus of this thesis. Despite the seeming synergy between functions that can be efficiently implemented and those that need to be, there still remain challenges in the area of polynomial datapath construction. Rewriting of polynomials in order to improve the results of logic synthesis is limited in industry standard logic synthesis tools. These synthesis tools are efficient at optimising the operations and precisions provided by the input hardware description language. They do not, however, cater for the exploitation of an
accuracy freedom, and as such do not perform lossy synthesis. Research into methods for exploiting an accuracy freedom typically do so without providing guaranteed statements on the nature of the error bring introduced and as such cannot be used in a controlled manner. These methods leave hardware designers having to perform functional testing on gate level netlists, an invariably expensive, time consuming and incomplete method of functional verification. The accuracy of floating-point algorithms are notoriously difficult to determine, this is also true of floating-point implementations of polynomials. There are applications where failure to implement polynomials with acceptable accuracy will result in catastrophic algorithm failure.

Such challenges are the focus of this thesis, moreover the nature of solutions found and presented in this thesis are designed to integrate into existing design flows in that they follow a high level synthesis paradigm. In this paradigm a high level statement of the design intent is transformed into a description encapsulated in a hardware description language (HDL) which is then subsequently synthesised by an industry standard logic synthesis tool. In particular, no changes to the logic synthesis is required. Industrial hardware design places necessarily stringent requirements on the formal verification of every transformation performed during the design process, as such methods for independently proving the correctness of the transformations constructed are sought throughout. In order to achieve these goals, the techniques used throughout the thesis are highly analytic, resulting in, where possible, HDL components that can be used off-the-shelf, enabling immediate use by hardware engineers. In this regard, this thesis succeeds in multiple aspects, providing HDL which has guaranteed bounded error properties and offers significant hardware implementation cost benefits. In certain cases, it has been possible to prove that the error freedom has been maximally exploited in minimising hardware resources. The thesis contains a variety of algorithms and procedures for optimising the implementation of fixed and floating-point polynomials, minimising resource usage while guaranteeing error properties. The method for ascertaining the quality of the approaches is by comparing the results of logic synthesis experiments.
1.1. Statement of Originality

A summary of the original contributions of this thesis per chapter are given here:

- Chapter 4 — a method for optimising the implementation of a set of mutually exclusive fixed-point polynomials with integer coefficients and a method for performing formal verification for designs which can be expressed as polynomials with integer inputs and coefficients via a super usage of industry standard tools.

- Chapter 5 — necessary and sufficient conditions for faithful rounding and worst case error vectors of constant correction truncated (CCT) and variable correction truncated (VCT) integer multiplication, a method for construction of the minimal faithfully rounded CCT and VCT integer multipliers, an optimal method for the construction of a faithful rounding of an arbitrary binary array, two new truncated multiplier architectures, a method for construction of the minimal multiply add schemes to perform constant division for round towards zero, round to nearest, even and faithful rounding.

- Chapter 6 — method for exploiting an arbitrary error when implementing an arbitrary binary array and constant division, a heuristic for assigning and exploiting a global bounded absolute error when implementing a fixed-point polynomial with rational coefficients for a given architecture and a hardware heuristic cost function for comparing the hardware area cost of implementing such polynomials.

- Chapter 7 — an algorithmic construction to place the framework put forward in Demmel et al. [46] into practice and a demonstration via a complete worked example as well as contributions towards full automation.

1.2. Impact

The following papers have been accepted for publication during the course of this thesis:
• **On The Systematic Creation of Faithfully Rounded Truncated Multipliers and Arrays**, Theo Drane, Thomas Rose and George A. Constantinides, IEEE Transactions on Computers, accepted for publication as a regular paper 5th June 2013.


• **Leap in the Formal Verification of Datapath**, Theo Drane and George A. Constantinides, published online on DAC.com Knowledge Center [48].


Aspects of this thesis have been presented at the following conferences:

• **Datapath Challenges** 21st IEEE Symposium on Computer Arithmetic (ARITH-21), 8th April 2013

• **Architectural Numeration** European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC), 22nd January 2013
• Harnessing the Power of Word-Level Formal Equivalency Checking, Verification Futures 2012, 19th November 2012

• Property Checking of Datapath using Word-Level Formal Equivalency Tools, Synopsys User Group (SNUG UK), 24th May 2012

• Correctly Rounded Constant Integer Division via Multiply-Add, IEEE International Symposium on Circuits and Systems (ISCAS), 22nd May 2012

• Formal Verification and Validation of High-Level Optimizations of Arithmetic Datapath Blocks, Synopsys User Group (SNUG Boston), 29th September 2011

• Formal Verification and Validation of High-Level Optimizations of Arithmetic Datapath Blocks, Synopsys User Group (SNUG France), 23rd June 2011

• Formal Verification and Validation of High-Level Optimizations of Arithmetic Datapath Blocks, Synopsys User Group (SNUG Germany), 19th May 2011

• Formal Verification and Validation of High-Level Optimizations of Arithmetic Datapath Blocks, Synopsys User Group (SNUG UK), 17th May 2011

• Formal Verification and Validation of High-Level Optimizations of Arithmetic Datapath Blocks, Synopsys User Group (SNUG San Jose), 30th March 2011

• Optimisation of Mutually Exclusive Arithmetic Sum-Of-Products, Design, Automation Test in Europe Conference Exhibition (DATE), 17th March 2011

The patents filed, so far, as a result of elements of this thesis are as follows:

• Method and Apparatus for Performing Synthesis of Division by Invariant Integers, US13/626886.

• Method and Apparatus for Performing Lossy Integer Multiplier Synthesis, GB1211757.8 and US13/537527.
• *Method and Apparatus for Performing the Synthesis of Polynomial Datapath via Formal Verification*, GB1106055.5 and US13/441543.


The commercial word-level formal equivalency checker SLEC from Calypto [22] has been improved by using the verification methodology presented in Chapter 4. The commercial word-level formal equivalency checker Hector from Synopsys [165] has been improved in order to prove the faithful rounding property of truncated multipliers. Internal tools have been developed at Imagination Technologies which implements the exploitation of an arbitrary error in the implementation of an arbitrary array.

Finally HDL, which derive from Chapters 4, 5 and 7, has been created and released to customers of Imagination Technologies.

1.3. Thesis Organisation

Chapter 2 presents background material on the logic synthesis of datapath components, high-level synthesis, lossy synthesis and the occurrence of polynomials within datapath. The manipulation of polynomials is a key element of the thesis, Chapter 3 presents concepts from algebraic geometry including an exposition on Gröbner bases and types of ideal membership testing which are used throughout the subsequent chapters. Chapters 4 through to 7 provide the technical contributions of the thesis. Hardware that supports multiple standards, floating-point datapath and general arithmetic logic units (ALUs) must implement mutually exclusive polynomials. Chapter 4 presents a method for optimising the implementation of a set of mutually exclusive polynomials with integer coefficients and inputs and an efficient method for performing formal verification of the transformation. Chapter 4 deals with a lossless transformation in that no accuracy freedom is exploited and demonstrates how best to implement a set of mutually exclusive polynomials by implementing a single polynomial with minimal control logic. Thus Chapter 4 can be seen as a preprocessing step that ultimately leads into requiring the implementation of a single polynomial. In Chapter 5 the components are created which ultimately lead to the creation of a lossy implementation of a fixed-point polynomial with rational coefficients.
These components include integer multipliers, multioperand addition and constant division operators. The method for creating a lossy multioperand adder leads to the creation of new truncated multipliers schemes. Optimal architectures are created which exploit an absolute error freedom and the analytic approach taken allows for these architectures to be directly embedded in HDL, allowing for off-the-shelf usage of these lossy components. A method is also presented in Chapter 5 which allows for the independent formal verification of these lossy components in order to establish their worst case error properties. Chapter 6 then uses the techniques put forward in Chapter 5 to first show how to construct an arithmetic sum-of-product (SOP) and constant division operator which optimally exploits an arbitrary absolute error bound to minimise implementation cost. Using these two types of lossy components a heuristic method is presented which explores how best to implement fixed-point polynomials with rational coefficients while exploiting a global error bound. In order to determine the quality of various architectures a hardware heuristic cost function is presented which attempts to produce a metric for the resultant area implementation cost. The final technical chapter deals with floating-point polynomial implementations. The fixed-point polynomial chapters dealt with exploiting, in a controlled manner, an absolute error bound; for floating-point it is natural to exploit a relative error bound. The only work found within the literature that appears to perform such a task is that of Demmel *et al.* [46]. Chapter 7 begins by first showing how the lossy integer multipliers of Chapter 5 can be used to create lossy floating-point multipliers with a guaranteed bounded relative error. This chapter then shows how such a component can be used to implement Demmel *et al.*’s vision by providing an algorithmic construction of Demmel’s framework and a complete worked example of the method on a Motzkin polynomial. This is followed by a discussion on the steps and hurdles towards complete generalisation. The thesis closes with Chapter 8 which summarises the contributions, their limitations and future work and directions.
2. Background

In 1965, Intel’s co-founder Gordon E. Moore made an important prediction which has arguably shaped as well as reflected miniaturisation within the semiconductor industry for the decades that followed. He predicted that the number of transistors packed into a single integrated circuit (IC) would grow exponentially, approximately doubling every two years [123]. Where ICs used to have thousands of transistors, they now number in the billions. For example, the recently introduced nVidia Kepler™GK110 has 7.1 billion transistors [127]. The transistor counts, clock speed, power and performance of Intel chip introductions can be found in Figure 2.1. Note the turning point, around 2003, where the power consumption dramatically ceased its exponential growth due to the difficulty of dissipating heat throughout a chip. A fixed power envelope with increasing transistor count has given rise to the notion of dark silicon [50], where power constraints would require the powering down of parts of chips or whole cores within a multicore system. The increase in transistor count provided the means for an increase in compute power and has enabled application specific chips to become an attractive proposition for general purpose computation [131]. An example of this is the rise of general purpose graphics processing units (GPGPUs) supporting the open computing language OpenCL [93]. This has only added to the plethora of changing standards modern chips must support. The increase in chip complexity has been accompanied by a shrink in time to market. A typical application specific integrated circuits (ASIC) design time is in the order of 12 to 18 months, which lags behind the 6 to 9 months required to develop products for the high consumer driven chip market [113]. This disparity is squeezing chip design time-lines, while the ratio of time spent between design and verification remains at a 30% to 70% ratio [140]. This thesis is written during a time when chips are being created where transistors are in greater abundance than the power to supply them and chips must be designed to support multiple standards in an ever
This chapter continues by providing background on the key areas that provide context to this thesis. The next section contains background on logic synthesis, the importance of datapath and the fundamentals of ASIC datapath design. High level synthesis (HLS) has long been seen as the answer to productivity challenges, the history of HLS and barriers to its adoption are presented in Section 2.2, which provide the challenges that the thesis overcomes such that the results can be used in practice. This is followed by Section 2.3 on the various existing approaches to lossy synthesis. Subsequently an exposition of the prominence of polynomials within datapath design is presented in Section 2.4. The chapter closes by summarising the context of the contribution in Section 2.5.
2.1. Logic and Datapath Synthesis

Chip complexity increasing in tandem with stringent design time constraints has pushed designers to embrace automation, abstraction and divide and conquer techniques [89]. This has given rise to the strong adoption of electronic design automation (EDA) at all levels of the design process, see Figure 2.2. Logic synthesis is the transition from a digital design at a register transfer level to gate level. The process explores the variety of possible implementations of logic functions given the desired design constraints, e.g. frequency, area and power. The foundations of logic synthesis date back to 1847’s Algebra of Logic created by George Boole [13] and Claude E. Shannon’s founding work on connecting this Boolean algebra with circuit design [148]. Having established the connection, Willard V. Quine went on, in the 1950s, to establish the minimisation theory of Boolean formulas in the two-level sum-of-products form [139]. A milestone in logic minimisation occurred in 1987 with the development of Espresso which had improved memory and computation runtime characteristics and became one of the core engines for the commercial logic synthesis tools that followed [142]. Around the same time, came the first instance of a commercial use of logic optimisation in

![Figure 2.2.: Levels of Design Abstraction.](image-url)
Table 2.1.: Commercial ASIC Synthesis Tools.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Vendor</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Compiler</td>
<td>Synopsys</td>
<td>[162]</td>
</tr>
<tr>
<td>Encounter RTL Compiler</td>
<td>Cadence Design Systems</td>
<td>[19]</td>
</tr>
<tr>
<td>RealTime Designer</td>
<td>Oasys Design Systems</td>
<td>[128]</td>
</tr>
<tr>
<td>BooleDozer</td>
<td>IBM (Internal)</td>
<td>[158]</td>
</tr>
</tbody>
</table>

Table 2.2.: Commercial LEC Tools.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Vendor</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>FormalPro</td>
<td>Mentor Graphics</td>
<td>[119]</td>
</tr>
<tr>
<td>Conformal</td>
<td>Cadence Design Systems</td>
<td>[20]</td>
</tr>
<tr>
<td>Formality</td>
<td>Synopsys</td>
<td>[164]</td>
</tr>
</tbody>
</table>

the guise of the task of remapping a gate level netlist from one standard cell library to another, Synopsys Remapper [89]. This period also saw IBM developing an in-house methodology where a simulation model expressed using a hardware description language (HDL) was used for logic synthesis [89]. This unified framework enabled the creation of a truly productive IC design methodology.

As logic synthesis automation developed and became commercially viable, the crucial task of proving the correctness of tool output became pressing, termed Logic Equivalence Checking (LEC). One of the early steps was the introduction of Binary Decision Diagrams (BDDs) [106] and the subsequent proof that a form of BDD, namely that of a reduced ordered BDD (ROBDD), provides a canonical representation [15].

Subsequent development has been heavily dominated by industrial design challenges, driven by the consistent exponential growth in circuit complexity. Issues such as tool runtimes, scalability, verifiability, supporting engineering change orders, attaining timing closure have all been heavily stressed by the sheer size of designs [89]. A set of commercial ASIC logic synthesis offerings can be found in Table 2.1. A set of commercial LEC tool offerings can be found in Table 2.2.

The synthesis of datapath circuits, as opposed to control path, is of vital importance. The percentage of logic and datapath within ICs continues to grow exponentially, see Figure 2.3. Datapath synthesis has challenged the precepts of logic synthesis and equivalence checking. Separate synthesis
Table 2.3.: Datapath Synthesis Tools and Libraries.

<table>
<thead>
<tr>
<th>Tool/Library</th>
<th>Vendor</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>DesignWare</td>
<td>Synopsys</td>
<td>[163]</td>
</tr>
<tr>
<td>CellMathDesigner</td>
<td>Forte Design Systems</td>
<td>[57]</td>
</tr>
<tr>
<td>FloPoCo</td>
<td>INRIA (academic)</td>
<td>[40]</td>
</tr>
</tbody>
</table>

Engines or libraries have been developed specifically targeted at datapath, see Table 2.3. Equivalence checking of arithmetic circuits has also required special attention [157], with arithmetic typically being the reason that full RTL to gate level equivalence fails to be achieved within design cycles. In truth, equivalence checking has failed to keep up with the exponential increase in chip complexity. The traditional flow of equivalence between RTL and gate level has had to be augmented by the logic synthesis tool passing hints to the equivalence checker which guides the tool to prove correctness, see Figure 2.4. This compromising of the integrity of equivalence checking proves the difficulty and importance of verification.

![Figure 2.3: Average Number of gates of Logic and Datapath, Excluding memories [59].](image)

Datapath synthesis rests on the literature of computer arithmetic. A host of architectures have been developed for the basic operations, e.g. multiplication, addition and division.
2.1.1. Integer Addition and Multiplication

Numeration, the art of the representation of numbers, ultimately plays an important role in the hardware costs of implementation, feasibility of particular architectures and even the ease and speed of debugging (it is not uncommon to encounter a situation where a complex algorithm specific numeration will be rejected in favour of a simpler, more easily maintainable alternative). Moreover, hardware implementations may benefit from changing the numeration throughout an architecture.

The most common of numerations is that of \( q \)-adic numbers, \( i.e. \) using radix-\( q \). Given that two-state logic pervades the chips, base 2 and powers of 2 have been by the far the easiest to implement. Notable exceptions include the Russian 1958 SETUN computer which used a \textit{balanced} ternary system using \(-1, 0 \text{ and } 1\) \cite{100}. Driven by the computational requirements of financial institutions, which in some cases require correctly rounded base 10 calculations by law, radix-10 arithmetic is a very active area of research \cite{171}.

The native number format supported by logic synthesis tools is that of binary integer. By interpreting a binary integer as having an implied binary point we have a fixed-point interpretation, \( i.e. \) \( m_{p-1}m_{p-2}\ldots m_1m_0 \times 2^e \) so for fixed integer \( e \) this scales the \( p \) bit number \( m \). So fixed-point operations reduce to integer operations. A survey of the architectures for integer addition and multiplication follow.
Integer Addition

A plethora of architectures exist for binary integer addition. To present the binary integer architectures it is useful to use the following notation:

- $\wedge$ logical AND
- $\vee$ logical OR
- $\oplus$ logical exclusive OR (XOR)

$s?a : b$ the ternary, if then else, operator

$a[i : j]$ bit slice of $a$ from columns $i$ down to $j$

The following functions aid in the presentation of the Boolean equation for $s_i$, a bit of $s = a + b$:

$$
g_i = a_i \wedge b_i \quad // \text{termed generate at position } i
$$

$$
p_i = a_i \vee b_i \quad // \text{termed propagate at position } i
$$

$$
t_i = a_i \oplus b_i
$$

$$
G_{i,j} = g_i \vee (p_i \wedge g_{i-1}) \vee (p_i \wedge p_{i-1} \wedge g_{i-2}) \vee \ldots
\vee (p_i \wedge p_{i-1} \wedge \ldots \wedge p_{j+1} \wedge g_j)
\quad // \text{adding } a[i : j] \text{ and } b[i : j] \text{ generates a carry}
$$

$$
D_{i,j} = g_i \vee (p_i \wedge g_{i-1}) \vee (p_i \wedge p_{i-1} \wedge g_{i-2}) \vee \ldots
\vee (p_i \wedge p_{i-1} \wedge \ldots \wedge p_{j+1} \wedge p_j)
\quad // \text{adding } a[i : j] \text{ and } b[i : j] \text{ and } 1 \text{ generates a carry}
$$

Then the general equation for $s_i$ is:

$$
s_i = t_i \oplus G_{i-1,0}
$$

The $G_{i-1,0}$ are the most complex terms to produce. The various adder architectures may be characterised by how they express the equation for $G_{i-1,0}$. The following are architectures whose delay is linear in the bit width of $a$ and $b$:

**Ripple Carry Adder** [17]  

$$
G_{i,0} = g_i \vee (p_i \wedge G_{i-1,0})
$$
Carry Skip Adder [107]

\[ G_{i,0} = G_{i,i-j} \lor (p_i \land p_{i-1} \land \ldots \land p_{i-j} \land G_{i-j-1,0}) \]

where \( G_{i,i-j} \) is constructed with a ripple carry architecture and this equation is recursed on \( i \) resulting in blocks of length \( j \) being processed.

Conditional Sum Adder [153]

\[ G_{i,0} = G_{i-j-1,0} \lor D_{i,i-j} : G_{i,i-j} \]

where \( G_{i,i-j} \) and \( D_{i,i-j} \) are constructed with a ripple carry architecture and this equation is recursed on \( i \) resulting in blocks of length \( j \) being processed.

The following is an architecture whose delay is logarithmic in the bit width of \( a \) and \( b \):

Carry Lookahead Adder [10]

\[
\begin{align*}
G_{i,l} &= G_{i,j} \lor (P_{i,j} \land G_{j-1,k}) \lor (P_{i,j} \land P_{j-1,k} \land G_{k-1,l}) \\
P_{i,l} &= P_{i,j} \land P_{j-1,k} \land P_{k-1,l} \\
\text{where} \quad P_{i,j} &= p_i \land p_{i-1} \land \ldots \land p_j \land p_j
\end{align*}
\]

Where recursion on \( G \) and \( P \) signals results in a logarithmic architecture. Ling adders are based upon the observation that \( g_i = p_i \land g_i \) and the creation of signals \( H_{i,j} \) such that \( G_{i,j} = p_i \land H_{i,j} \). This results in the following logarithmic architecture:

Ling Adder [111]

\[
\begin{align*}
H_{2,0} &= g_2 \lor g_1 \lor (p_1 \land g_0) \\
H_{i,l} &= H_{i,j} \lor (P_{i,j-1,k} \land H_{j-1,k}) \lor (P_{i,j-1} \land P_{j-2,k-1} \land H_{k-1,l}) \\
s_i &= H_{i-1,0} \lor t_i \oplus p_{i-1} : t_i
\end{align*}
\]

Recursion on \( H \) and \( P \) signals results in a logarithmic architecture. The Ling adder observation that \( p_i \) is a factor of \( G_{i,j} \) was generalised to the observation that \( D_{i,k} \) is a factor of \( G_{i,j} \) for \( i \geq k \geq j \), this gives rise the the Jackson adder:
Jackson Adder [85]

\[ B_{i,k} = B_{i,j} \lor B_{j-1,k} \quad P_{i,k} = P_{i,j} \land P_{j-1,k} \]
\[ G_{i,k} = G_{i,j} \lor (P_{i,j} \land G_{j-1,k}) \quad G_{i,k} = D_{i,j} \land (B_{i,j} \lor G_{j-1,k}) \]
\[ D_{i,k} = G_{i,j} \lor (P_{i,j} \land D_{j-1,k}) \quad D_{i,k} = D_{i,j} \land (B_{i,j} \lor D_{j-1,k}) \]

where \( B_{i,j} = g_i \lor g_{i-1} \lor \ldots \lor g_{j+1} \lor g_j \)

Where the recursions on \( D, G, P \) and \( B \) allow the creation of a logarithmic architecture, note that a plethora of architectures are possible as at each level of the recursion there is a choice of decomposition for \( G \) and \( D \).

A survey can be found in [56]. Modern adder implementations use a hybrid of all of these architectures and logic synthesis uses a timing driven approach in creating these hybrids. The logarithmic adder constructions typically fall into a class known as parallel prefix, a full exploration of such adder types can be found in [149].

**Integer Multiplication**

For binary integer multiplication the standard architecture for fully parallel implementations is that of array creation, array reduction and final integer addition. The array construction of the partial products normally falls into two types, an AND array which is the result of the simplest add-and-shift multiplication algorithm and Booth arrays which have fewer partial products at the expense of some encoding logic of the multiplier input.

The AND array is based upon the following mathematical formulation for two \( n \) bit binary inputs \( a \) and \( b \):

\[
y = ab = \left( \sum_{i=0}^{n-1} a_i 2^i \right) \left( \sum_{j=0}^{n-1} b_j 2^j \right) = \sum_{i,j=0}^{n-1} a_i b_j 2^{i+j}
\]
\[
= a_i \text{ AND } b_j
\]

Figure 2.5.: Structure of AND Multiplication Array.

The array of bits formed when \( n = 4 \) is then:

\[
\begin{array}{ccccccc}
2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
\end{array} \\
\begin{array}{ccccccc}
a_0b_3 & a_0b_2 & a_0b_1 & a_0b_0 \\
a_1b_3 & a_1b_2 & a_1b_1 & a_1b_0 \\
a_2b_3 & a_2b_2 & a_2b_1 & a_2b_0 \\
a_3b_3 & a_3b_2 & a_3b_1 & a_3b_0 \\
\end{array}
\]

Each row is called a *partial product*. Summing all the partial products will produce the multiplication result. It is common to represent the multiplication array as a dot diagram. The structure of a general AND array can be found in Figure 2.5.

The most common form of Booth array is that of Booth radix-4 which slices the multiplier input into overlapping triples [14]. It is based upon the following mathematical formulation for two \( n \) bit binary inputs \( a \) and \( b \):

\[
y = ab \\
= a \left( \sum_{j=0}^{n-1} b_j 2^j \right) \\
= a \left( \sum_{j=0}^{\lfloor \frac{n}{2} \rfloor} (-2b_{2j+1} + b_{2j} + b_{2j-1}) 2^{2j} \right) \\
= \sum_{j=0}^{\lfloor \frac{n}{2} \rfloor} [a (-2b_{2j+1} + b_{2j} + b_{2j-1})] 2^{2j}
\]
Now note that $-2b_{2j+1} + b_2 + b_{2j-1} \in \{-2, -1, 0, 1, 2\}$ for all the possible values of the bits $b_{2j+1}$, $b_2$ and $b_{2j-1}$. So this formulation requires the creation of the following multiples $-2a$, $-a$, 0, $a$ and $2a$. All these multiples are easy to produce in a redundant representation of the form $pp[n+1 : 0] + s$, where the most significant bit of $pp$ requires negation and $s$ is a one bit variable:

\[
\begin{array}{cccccccccc}
2^{n+1} & 2^n & 2^{n-1} & 2^{n-2} & \ldots & 2^1 & 2^0 \\
p_{n+1} & p_n & p_{n-1} & p_{n-2} & \ldots & p_1 & p_0 \\
-2a &=& -a_{n-1} & a_{n-2} & a_{n-3} & \ldots & a_0 & 1 \\
-a &=& -a_{n-1} & a_{n-1} & a_{n-2} & \ldots & a_1 & a_0 & 1 \\
0 &=& -0 & 0 & 0 & 0 & \ldots & 0 & 0 & 1 \\
a &=& -0 & 0 & a_{n-1} & a_{n-2} & \ldots & a_1 & a_0 & 0 \\
2a &=& -0 & a_{n-1} & a_{n-2} & a_{n-3} & \ldots & a_1 & a_0 & 0 \\
\end{array}
\]

The expansion required for $n = 10$ is:

\[
y = a(-2b_1 + b_0) + 2^2a(-2b_3 + b_2 + b_1) + 2^4a(-2b_5 + b_4 + b_3) + 2^6a(-2b_7 + b_6 + b_5) + 2^8a(-2b_9 + b_8 + b_7) + 2^{10}a(b_9)
\]

Note that the first and last partial products are of a simpler form than the others, in particular, the last partial product is never negative so no $s$ partial product bit is required. Hence a Booth radix-4 array for $n = 10$ can be created as in Figure 2.6. The hollow circles require negation, this negation can be achieved by inverting these bits and adding ones into the array as in Figure 2.7. Finally the array height can be reduced by noting
that the top of the array can be simplified by combining bit $A$ with some of the constant ones. The final Booth radix-4 array can be found in Figure 2.8. The arrays vary slightly in their construction and complexity per partial product bit depending on whether the inputs are signed or unsigned and whether $n$ is even or odd.

Booth radix-4 requires approximately $n/2$ partial products as opposed to the $n$ required by an AND array. This reduction comes at the cost of creating the more complex partial product bits, however the benefits outweigh the costs, making Booth radix-4 one of the commonly found architectural alternatives to an AND array. There are other Booth type schemes which require non power of 2 multiples of the multiplicand, this invariably means that the cost of array construction outweighs the benefit from the further reduction in number of partial products.

Having constructed the array, the next step is to reduce the array. Array
reduction is typically performed by repeated use of small reduction cells such as full-adders and half-adders which sum three and two bits respectively of equal binary weight. An example reduction of an AND array for \( n = 5 \) can be found in Figure 2.9.

![Array Reduction of 5 bit AND Array Multiplier.](image)

A whole range of generalised parallel counters have been put forward [118]. In particular, efficient and logarithmic methods have been found for designing single reduction cells that implement the sum of \( n \) bits. Compressor cells are also used, these take bits from different columns and partially reduce them, a common example is the 4 to 2 compressor (functionally identical to 2 full adders). The array reduction phase reduces the array to a height of two. The allocation of reduction cells used to perform the reduction is known as a topology, of which there are regular and irregular forms. Regular topologies ease the task of multiplier layout and ignore logic delay through the reduction cells, whereas irregular topologies attempt to reduce logic delay regardless of the effect on layout. Wallace tree reduction is timing driven and greedy in the sense that as much reduction is performed as possible in every part of the multiplier array [176]. Dadda trees are also irregular multiplier topologies but do as little as possible when reducing the parts of the array furthest from the region with maximum height [34]. The reduction of the array is continued until the array is of height two, at which point a final binary integer adder is used. By using a conditional sum approach, the adder can be molded to fit the output delay profile of the array reduction phase. There are a myriad of hybrid schemes based upon
the fundamental architectures presented here [56].

2.1.2. Datapath Synthesis Flow

Typically, arithmetic operators found within the RTL will first be grouped into datapath blocks. The arithmetic binary integer operations typically grouped are:

- $A \pm B \pm C \ldots$ multioperand addition and subtraction
- $A \times B, A \times A$ multiplication and squaring
- $A == B, A > B$ equality checking and comparisons
- $A >> B, A >>> B$ shifters
- $A \times B \pm C \times D \pm E \ldots$ sum-of-products (SOP)
- $(A + B) \times C, (A \times B) \times C$ product-of-sums
- $S?A : B : C : D$ selection or muxing

Optimisations are then performed on these expressions, an example of such an optimisation is $A(B + C) = AB + AC$. Finally, these optimised expressions are then mapped to gates through dedicated datapath generators. When constants appear on some of the inputs, efficient algorithms exist to exploit these circumstances, for example multiplication by a constant [132]. Crucially, the choice of optimisations and architectures used in the mapping is driven by hardware constraints such as arrival times of the inputs to the datapath block, area, power and timing constraints as well as properties of the standard cell library in use [183].
2.2. High-Level Synthesis

High-Level Synthesis (HLS), also known as algorithmic, behavioural or electronic system level synthesis, automates the design process which interprets a system level model and produces a register transfer level description. The benefits of such an approach are [54]:

- **Architectural Exploration** — the productivity gained by automating the creation of RTL, rather than hand coding its creation, allows the designer to explore multiple design choices quickly. Time to market may mean that the rush is to produce a single functioning design where there is barely time for complete verification, let alone exploring multiple architectures. The lack of time to explore architectures stunts designers’ ingenuity and limits innovation.

- **Coping with Complexity** — the system level code that is equivalent to a piece of RTL is invariably an order of magnitude smaller in length. Designers can own larger amounts of chip design, thus handling the increasing design complexity. Having fewer people who intellectually understand the entire architecture enables greater architecture exploration and reduces the risk of bugs.

- **Reducing Design and Verification Effort** — the RTL is automatically generated from a smaller and simpler system level model, thus eliminating the possibility of hand introduced RTL bugs.

- **Enabling Reuse** — RTL specifies the cycle accurate behaviour of the design, if the underlying process technology changes the register boundaries may need moving. Performing this by hand is error prone and tedious. HLS forces a separation of duty onto the designer — functionality on the one hand and control (level of parallelism, scheduling etc.) on the other. This separation allows for automatic re-targeting of the design to a new process.

- **Investing Research and Development Where It Matters** — the productivity gain allows for more algorithm development and architectural optimisations. Without HLS, the race is towards RTL existence, with HLS, the challenge is to create optimised RTL.
The history of HLS tool development can be split into three generations [117]. The first generation of HLS tools arose during the 1980s and early 1990s. Highlights of this period was work emerging from Carleton University and IMEC. The latter produced Cathedral and Cathedral-II and focused on digital signal processing (DSP) algorithms and used a domain specific language (DSL) [42]. The commercialisation of this tool ultimately failed but not before passing through the hands of Mentor, Frontier Design, Adelante and ARM. Generation 1 failed for four reasons [117]: need — the industry was only just adjusting to the adoption of RTL synthesis, language — industry adoption of an academic DSL is unlikely, quality of results (QoR) — the tools were too primitive to offer competitive QoR and finally specificity — the tools only worked on a very limited domain.

The second generation, mid 90s to early 2000s saw the leading EDA tool vendors release their own HLS offerings, the most notable of which was Synopsys Behavioral Compiler [99]. Generation 2 failed for the following reasons: synthesising all the way down to gates — the HLS tools did not complement the existing flows but replaced them with worse QoR, wrong input language — the input was behavioural HDL which competed with the existing RTL flow and did not raise the abstraction level such that compiler based language optimisations could be used, poor QoR — this was particularly evident in control flow, validation — no formal methods existed to prove equivalence of the system level model with the resultant RTL, simulation runtimes — simulation runtimes of the behavioural system level model were almost as long as those of the HDL.

The third generation, running from the early 2000s to the present, is gaining traction within the industry. The success has come from the following factors: appropriate domain marketing — most modern tools are dataflow and DSP design focused, language — HLS tools are now using C, C++ or SystemC [2] which is finally a familiar abstraction level above RTL, improved QoR — improvements have come from compiler based optimisations, shift in user designs — RTL designers are having to deal with an increasing range of potentially unfamiliar applications, thus those encountering signal or multimedia processing are reaching for HLS tools.

A list of some of the current HLS offerings and their properties can be found in Table 2.4. Challenges for these tools are:
Table 2.4.: High-Level Synthesis Tools.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Vendor/University</th>
<th>Logic Synthesis</th>
<th>Floating Point Support</th>
<th>Sequential Equivalence Checking</th>
<th>ECO</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthesizer</td>
<td>Forte Design Systems</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>[58]</td>
</tr>
<tr>
<td>C-to-Silicon</td>
<td>Cadence Design Systems</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>[18]</td>
</tr>
<tr>
<td>CatapultC</td>
<td>Calypto Design Systems</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>[21]</td>
</tr>
<tr>
<td>BlueSpec</td>
<td>Bluespec, Inc.</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>[12]</td>
</tr>
<tr>
<td>Symphonic</td>
<td>Synopsys</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>[166]</td>
</tr>
<tr>
<td>Vivado</td>
<td>Xilinx, Inc.</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>[179]</td>
</tr>
<tr>
<td>GAUT</td>
<td>Uni. of South Brittany</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>[116]</td>
</tr>
<tr>
<td>LegUp</td>
<td>Uni. of Toronto</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>[169]</td>
</tr>
</tbody>
</table>

• **Sound Architectural Exploration** — HLS tools must be able to evaluate the quality of a particular architectural choice. To do so, they must interface, potentially iteratively, with a logic synthesis tool. Further, they must be able to correctly ascertain whether or not a target frequency can be met by a given architecture. To do so requires tight integration with proven logic synthesis tools. However, tools such as CatapultC have no such integration. Such tools use a database of area, delay and power figures for the fundamental operators, such as integer multiplication, for a given technology library to ascertain the expected delay of datapath. This would build an architecture that implemented integer $x_1 + x_2 + \ldots + x_n$ out of two input integer adders, where in fact such datapath would be optimised by leading logic synthesis tools into a single partial product array. HLS tools which schedule at an operator level and which do not use logic synthesis tools in its architectural explorations will fail even to provide the most basic of architectures an RTL engineer would use. HLS tools with access to logic synthesis tools are indicated in Table 2.4. Hence the quality of an HLS tool is dependent on the existence of a logic synthesis partner.
• **Number Format Support** — in addition to arbitrary sized signed and unsigned integer datatypes, HLS tools also support arbitrary sized signed and unsigned fixed-point types via the use of SystemC. This is a slightly useful abstraction gain compared to integer types found in HDL. Full C/C++/SystemC support requires support for the floating-point datatype. HLS tools wishing to provide good floating-point support will either use existing libraries of optimised floating-point components or invest in creating their own. HLS tools which have floating-point support are indicated in Table 2.4.

• **Verification** — with automation comes loss of trust, designers must know that the tool output functions as expected. This requires sequential equivalence checking (SEC) between the C/C++/SystemC and the resultant RTL. The only commercial product which supports HLS tools is Calypto’s SLEC [22]. Setting up the notion of equivalence between the untimed system level model and the timed RTL can be non trivial due to the ways that an HLS tool may be driven in terms of controlling throughput and latency. This complexity has meant that some HLS tools produce scripts for the SEC tools. Designers will have to inspect these scripts and potentially deal with the problem of the SEC not attaining a proof of equivalence. Verification, in general, is a crucial barrier to greater HLS adoption.

• **Engineering Change Orders (ECO)** — these occur when bugs are discovered late into the design cycle, at the point where only the smallest change to a netlist is permitted. Thus an HLS tool must be able to be run in *ECO mode* where the system level input is fixed and the tool must produce the equivalent RTL which hopefully has only localised combinatoric changes compared to the original. Given the high level nature of the tool, this is a serious challenge which must be overcome. The tools with ECO support are highlighted in Table 2.4.

The excitement of HLS should not be based upon productivity gains but the optimisations that the traditional design process cannot hope of achieving. HLS’s benefit from compiler optimisations such as dead code removal, common sub expression extraction, associativity and distributivity optimisations can now be brought to bear on RTL designs. However
compiler optimisations need rethinking in light of HLS needs, for example dealing with mutually exclusive operators [31]. One important area where automation is crucial is that of bit width analysis [27]. RTL engineers must consider which bit widths to use for every signal and unless formal verification is used, simulation is unlikely to find bugs introduced by incorrect widths. Some HLS tools can establish necessary bit widths by forward and backward propagating interval arithmetic. More work needs to be done to raise bit level concerns to the height of being used by HLS tools, for example, multiplier fragmentation where integer multipliers can be fragmented into smaller multipliers. Ultimately HLS should be aiming at real architectural exploration, for example, being able to move between differing radices of Fast Fourier Transforms.
2.3. Lossy Synthesis

Lossy synthesis is a method of synthesis where the result is not bit identical to the input. The user specifies and controls the error and the lossy synthesiser exploits the error freedom to optimise hardware area, speed and power [28]. Exploiting the error freedom requires tweaking particular parameters within the circuits to be created. This opens up a staggering array of options and research has been performed at various levels of the design process. Three of the approaches are word length optimisation, imprecise operators and gate level imprecision.

2.3.1. Word-length Optimisation

The word-length, i.e. the number of bits, used to represent internal variables within a design depend on the range and precision of data the variable must hold, as well as the number format. The precision of every signal within a design contributes to the overall error and hardware properties. Choosing the best choice of precisions for all internal variables from hardware considerations while maintaining acceptable global error is a form of lossy synthesis. This has been shown to be NP-hard [30] in general.

In practice, it is common that a design may be extensively simulated to confirm whether the design is numerically sound, both in terms of range and precision, e.g. [103]. However this approach fails to give real confidence in the design due to the limited number of simulations and this approach scales poorly as the precisions of the inputs grow. More importantly, this approach fails to provide any intellectual understanding of the design at hand.

More formal approaches for range analysis include the use of interval arithmetic and affine arithmetic, however these suffer from not providing tight bounds [124], [51] and [129]. These have been augmented to include the iterative use of Satisfiability Modulo Theories (SMT) [9] which provide tight bounds but with the risk of the computational complexity resulting in the SMT solvers being unable to provide proofs in reasonable time. However their use is iterative with a bound being produced at each stage which becomes refined, thus early termination still provides a bound [98]. Another approach is to use arithmetic transforms (AT) which convert the circuit at
hand into a polynomial of the following form:

\[
\sum_{i_1=0}^{1} \sum_{i_2=0}^{1} \cdots \sum_{i_n=0}^{1} c_{i_1,i_2,\ldots,i_n} x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}
\]

where \( x_1, x_2, \ldots, x_n \) are bits and \( n \) is the total number of input bits to the design. The coefficients \( c \) are integer and the output is an integer representation of the output word of the circuit. Note that every circuit can be expressed in this way, every boolean function can be expressed as a composition of NAND gates and the functionality of a NAND gate can be expressed as the polynomial \( 1 - ab \) where \( a \) and \( b \) are the inputs of the NAND gate. Composing these polynomials will result in a polynomial with integer coefficients. Note that for any binary variable \( a^n = a \), so the polynomial can always be simplified such that the power of any binary input never exceeds one. If \( p_i \) is the polynomial representing the boolean equation for bit \( i \) in the output word \( y \) then:

\[
y = p_0 + 2^1 p_1 + 2^2 p_2 + \ldots + 2^i p_i + \ldots
\]

A sum of polynomials with integer coefficients is a polynomial with integer coefficients, thus an AT representation exists for all circuits. By performing a binary branch and bound on the AT the range of the output can be determined [133]. These bounding techniques can be used to establish range and whether the absolute global error is sufficiently bounded. In the case where the error variance needs controlling it is typically assumed that every rounding event introduces an independent source of uniformly distributed error [29], [55] and [5].

The circuit cost can be measured by performing the logic synthesis and performing power estimations on the resultant gate level netlist. However, this is time consuming and given that many architectures may need exploring a typical approach is to create an heuristic that captures the hardware properties of interest. Heuristics have included the number of terms in an AT as giving a strong indication of associated hardware cost [133] and using gate counts to estimate the effect of changing precision [29].

In terms of navigating the solution space the approaches that have been taken include genetic algorithms [129], [62] and [5]. An iterative algorithm inspired by a steepest descent method has been used in [55]. Using cost
heuristics, the lossy synthesis optimisation has been phrased as a mixed integer linear program [29] which is suitable for small problem sizes. A heuristic approach which steps through the space of possible precisions has been presented in [62], [29] and [133]. In [62] and [29] the starting point is the smallest uniform precision. To provide information about the direction in which to move a notion of sensitivity of the output to a given internal precision has been used, firstly via performing differentiation on the polynomial representing the design in [133] and secondly via automatic differentiation [74] in [62].

### 2.3.2. Imprecise Operators

In 2012 a chip was built that was up to 15 times more efficient by allowing a deviation of up to 8% in numerical output accuracy [112]. This was achieved by altering the functionality of integer adders and multipliers within the chip. The approach was to relinquish the conventional wisdom that the fundamental operators need to return the *correct* answer. An imprecise approach to operator correctness is permissible when the applications involved can naturally tolerate errors, for example image processing.

Considering the integer adder, in the case of a ripple carry adder a carry needs to ripple all the way from the least significant to most significant bit. Delay, area and power gains can be achieved by simply cutting the carry chain. The idea is that long carry chains are unlikely in practice but yet determine the circuit complexity, disposing of this unlikely circuitry should result in a design with a low probability of large errors. A *sloppy* adder has been proposed in [6] with the following imprecise functionality for an addition of \( a[n - 1 : 0] \) and \( b[n - 1 : 0] \) with \( k \) least significant *sloppy* bits:

\[
s[k - 1 : 0] = a[k - 1 : 0] \oplus b[k - 1 : 0] \\
\]

\[
s[n : k] = a[n - 1 : k] + b[n - 1 : k] \\
\]

The ETAIIM adder (error tolerant adder type II modified) is an imprecise version of the carry skip adder where carries are really skipped to cut the
carry chain, its functionality is as follows:

\[
\begin{align*}
 s'[k - 1 : 0] &= a[k - 1 : 0] + b[k - 1 : 0] \\
 s'[2k - 1 : k] &= (a[2k - 1 : 0] + b[2k - 1 : 0]) \gg k \\
 s'[3k - 1 : 2k] &= (a[3k - 1 : k] + b[3k - 1 : k]) \gg k \\
 s'[4k - 1 : 3k] &= (a[4k - 1 : 2k] + b[4k - 1 : 2k]) \gg k \
\end{align*}
\]

\[\ldots\]

\[
 s'[n - 1 : mk] = (a[n - 1 : (m - 1)k] + b[n - 1 : (m - 1)k]) \gg k
\]

The least significant portion of the adder is sliced into \( m \) regions, each of length \( k \). The carry chains in the least significant region are limited to a length of \( 2k \), whereas the most significant portion is computed correctly using a carry from the least significant portion [180]. The almost correct adder (ACA) is an imprecise variant of a parallel prefix adder and ignores the last few levels of the logarithmic carry generation [175]. It was noted in [82] that all of the imprecise schemes essentially insert zeroes into the carry chain at specific points and thus have biased errors. Instead of setting \( G_{i,0} = 0 \) they propose using \( G_{i,0} = a_i \) or \( G_{i,0} = b_i \) in an attempt to reduce the asymmetry in the error.

For imprecise integer multipliers [82] proposed only replacing the final binary addition by an imprecise adder and left the rest of the architecture unchanged. In [6] the Booth radix-4 multiplier architecture was altered during the array creation. Typically each partial product bit of a Booth radix-4 array is a function of three bits of \( a \) and two bits of \( b \), [6] consider two ways of simplifying this logic, only using 2 bits of \( a \) and/or only one bit of \( b \) and applying this to one or both of the least significant columns and rows. Other work has concentrated on the array reduction and designed a range of simplified full adder cells [75].

These imprecise components already provide a considerable number of fundamentally functionally different multiplier and adder architectures and any one of these could be used for any of the instances within the design. In [82] 39 adder designs and 101 multiplier designs are considered by using a variety of imprecise components.

Research into these imprecise operators has provided analytic formulae for the error properties of the sloppy full adder [75] and some of the impre-
cise adders. At a design level the effect of the imprecision has been captured by simulation of use cases and taking a measure such as peak signal to noise ratio [75]. A more analytic approach has been taken in [82] where interval arithmetic was used to propagate probability density functions through designs consisting of only fixed-point adders and multipliers to determine output probability distribution.

A variety of approaches has been taken to the optimisation problem of how to use these imprecise components to optimise certain hardware properties while maintaining an acceptable level of output error. In [81] experiments were performed to establish the error, power and delay of a set of ACAs, each having a different number of logic levels. Convex curves were fitted to the experimental results, which allowed for the formulation of a convex optimisation problem to solve the lossy synthesis problem. An analytically tractable problem was also formulated in [75] which was enabled by creating a heuristic for the power of a ripple carry adder made from sloppy full adders. In [82] a genetic algorithm was combined with power runs for hardware quality and interval arithmetic for error quality.

2.3.3. Gate Level Imprecision

Word-length optimisation and the use of imprecise operators are manifestations of the simple idea that there may be logic gates that can be removed from a circuit while still maintaining some level of quality. This is the idea behind SALSA (Systematic methodology for Automatic Logic Synthesis of Approximate circuits) [172] and SASIMI (Substitute-And-SIMplIfy) [173]. These tools require the creation of a circuit whose output bit is high if the approximate circuit has acceptable quality. For example, if the original circuit was $y_{\text{orig}}(x_1, x_2, ..., x_n)$ and a candidate approximate circuit is $y_{\text{approx}}(x_1, x_2, ..., x_n)$ and quality can be defined via an absolute error constraint then the functionality of the circuit required is:

$$s = Q(y_{\text{orig}}, y_{\text{approx}})$$

$$= (|y_{\text{orig}}(x_1, x_2, ..., x_n) - y_{\text{approx}}(x_1, x_2, ..., x_n)| < E)?1 : 0$$

where $Q$ is the quality function. Gates within $y_{\text{approx}}$ can be manipulated as long as $s = 1$ for all allowed values of the inputs $x_1, x_2, ..., x_n$. SALSA first finds the outputs of $y_{\text{orig}}$ and $y_{\text{approx}}$ such that $Q$ is unaltered if the
ith bit of $y_{approx}$ is either one or zero. Then by passing external don’t cares to a logic synthesis tool, the circuit for $y_{approx}$ can be optimised by using this freedom. This can be done for each output bit of $y_{approx}$. SASIMI tries to find internal points which compute the same value with high probability and substitutes one for the other, this process is performed iteratively while maintaining the fact that the quality function always evaluates to one.
2.4. Polynomials in Datapath

Polynomials are commonly found within datapath. Recall from Section 2.1.2 that the set of operators extracted during datapath synthesis is:

- $A \pm B \pm C \ldots$ multioperand addition and subtraction
- $A \times B, A \times A$ multiplication and squaring
- $A \times B \pm C \times D \pm E \ldots$ sum-of-products (SOP)
- $(A + B) \times C, (A \times B) \times C$ product-of-sums
- $S?A : B : C : D$ selection or muxing
- $A >> B, A >>> B$ shifters
- $A == B, A > B$ equality checking and comparisons

Compositions of the first four of these types of operations will obviously result in polynomials. In the case of muxing, this is also polynomial in nature:

$$S?A : B : C : D = (1 - S_1)(1 - S_0)A + (1 - S_1)S_0B + S_1(1 - S_0)C + S_1S_0D$$

Moreover left shifting is also expressible as a polynomial:

$$A << B[n - 1 : 0] = A \left( \left( 2^{2^n-1} - 1 \right) B_{n-1} + 1 \right) \ldots (15B_2 + 1)(3B_1 + 1)(B_0 + 1)$$

Certain logical operations are also polynomial:

- $\overline{a[n - 1 : 0]} = 2^n - 1 - a$ // inversion
- $a[n - 1 : 0] \oplus s = (1 - 2s)a + s(2^n - 1)$ // XORing each bit of $a$ with bit $s$

Hence methods for optimising polynomials with integer inputs and coefficients are of relevance to datapath synthesis engines.

A variety of number formats have polynomial interpretations. The interpretation of $x[n - 1 : 0]$ as one of a variety of number formats can be written as polynomials with integer inputs and rational coefficients:

- Fixed-Point $\frac{x}{2^m}$
- Signed $-2^{n-1}x_{n-1} + x[n - 2 : 0]$
- Sign Magnitude $(1 - 2x_{n-1})x[n - 2 : 0]$
Certain graphics formats can also be written as such:

\[ \begin{align*}
\text{UNORM} & : x[n-1:0] \\
\text{SNORM} & : -\frac{2^{n-1}x_{n-1} + x[n-2:0]}{2^{n-1}-1}
\end{align*} \]

Floating-point formats, ignoring exceptions, can also be written in a polynomial form. If the floating-point format is the concatenation of sign \( s \), exponent \( e \) and mantissa \( m \) where the exponent and mantissa are of width \( ew \) and \( mw \) respectively, then the interpretation can be written as:

\[
(-1)^s 2^{(e-2^{ew-1}+1)} 1 \cdot m = \frac{(1 - 2s) \left( 2^{ew-1} - 1 \right) e_{ew-1} + 1 \ldots (3e_1 + 1)(e_0 + 1) (2^{mw} + m)}{2^{2^{ew-1}+mw-1}}
\]

So polynomial operations involving any of these formats will require the implementation of polynomials with integer inputs and rational coefficients.

More complex operations such as function evaluation, \( e.g. \) \( \sin(x) \), typically use fixed-point polynomials to perform part of the evaluation. Finally, at the application level, polynomials with rational coefficients are found in countless domains, media processing, graphics, communications which require implementations of algorithms from signal processing, linear algebra, filtering, \textit{etc.}

In summary, polynomials can be used to express the functionality of a significant portion of datapath design as well as being directly required for the implementation of a large number of algorithms.
2.5. Context of Contribution

Lossy synthesis has the potential to overcome the power consumption challenges facing chip design. Circuits producing results of varying accuracy and power may be desirable due to having to support multiple standards or working in applications which have an inherent error tolerance. Power restrictions but with increased transistor counts may permit the creation of separate circuits for each accuracy or logic sharing. Imprecise operators have shown benefit beyond word-length optimisation but they have not been designed to exhibit particular error properties, they are treated as black boxes with particular parameters that can be altered. Further, imprecise operators have not been explored for datapath blocks which are a combination of fundamental operators, such as sum-of-products. Formal proof of correctness of imprecise circuits and the optimal use of error freedom is crucial to enable adoption of assert quality of lossy circuits. The implementation of polynomials with rational coefficients is a common requirement.

For all of these reasons, this thesis contributes techniques towards the realisation of a lossy synthesiser of the form shown in Figure 2.10. This synthesiser takes as input a polynomial expressed symbolically and the formats of the inputs and output with their associated precisions. More precisely polynomials with rational coefficients with fixed-point inputs and outputs will be considered as well as polynomials with floating-point coefficients and floating-point inputs and outputs. Producing a result which is infinitely precise is typically impossible with a finite precision output, thus a notion of acceptable error must be provided. In the case of a fixed-point output the error will be considered to be acceptable if its absolute value is bounded by a user defined limit. In the case of a floating-point output, acceptable error will be a bounded relative error whose limit is user defined. The output is RTL that is guaranteed to meet the error specification, is intended to optimally exploit the error specification and should be suitable for datapath logic synthesis.

In order to guarantee that the error specification is met, the error properties of the architectures considered are analysed analytically. Where possible, independent formal verification techniques are created which prove the correctness of the resultant RTL. The architectures used to meet and exploit the error specification will be imprecise datapath operators. These
methods go beyond word-length optimisation by using imprecise datapath operators but in order to use these components their error properties have to be analytically investigated. An arithmetic sum-of-product operator will be a fundamental building datapath block used throughout the thesis. The method of exploiting the error specification considered is that of the removal of partial product bits that form the array of the sum-of-product operator. Where possible, proofs are provided that show that the error has been maximally exploited, such that for a given architecture, there can be no better implementation that still meets the error specification.

The use of the mathematical description of the polynomial as the input to the lossy synthesiser is at a higher abstraction level that is typically considered by HLS tools. This form of input is closer to the designers’ intent and places the burden of establishing internal precisions on the tool, but does require a user defined error specification. This will need to be derived from an application level understanding of acceptable error. A typical user requirement is the result should be correct to the unit in the last place, for a fixed-point output this can be translated into an absolute error bound and for floating-point this can be translated into a bound on absolute value of the relative error. Thus the challenge for the user is to define the output precision. This architectural choice is ultimately determined by a notion of application level quality and an associated fidelity metric. For example audio, image and video decompression have an application level quality which is subjective human perception and is translated into a signal to noise ratio which in turn is used to determine internal precisions. Other
examples include lossy decompression algorithms, artificial intelligence applications where current states of belief are held in probability distributions [110] and situations where standards specifically state error freedom when implementing complex functions such as $\sin(x)$ [93]. This thesis uses the inherent application level error freedom to reduce the hardware implementation costs by design time architecting. Application level error freedom can also be used to recover from real-time faults and perform real-time performance improvements such as voltage scaling [110], [147].
3. Preliminaries

The manipulation of polynomials plays a central role in the chapters that follow. Advances in computational algebraic geometry not only enable automated and efficient implementations of the algorithms presented in this thesis, but also their continued development will naturally increase the scope of designs that are tractable by such algorithms. This chapter presents background material on polynomial rings, Gröbner bases and ideals. This material is derived from [33], [4], [73] and [32]. The motivating questions answered by these preliminaries are:

Q1 Building Blocks Can a given polynomial be written as a polynomial function of a set of others? For example can the polynomial:

\[(ac - bd)^2 + (ad + bc)^2\]

be written as a polynomial in \(u\) and \(v\) where \(u = a^2 + b^2\) and \(v = c^2 + d^2\)?

The this case, the answer is yes and the decomposition is known as the Brahmagupta-Fibonacci identity\(^1\):

\[(ac - bd)^2 + (ad + bc)^2 = (a^2 + b^2)(c^2 + d^2)\]

Q2 Verification over the Finite Field Equality of polynomials is more complex when dealing with finite arithmetic, for example consider these two designs:

\[y_1[2 : 0] = x^4 + 2x\]
\[y_2[2 : 0] = 2x^3 + x^2\]

where the least significant three bits of these two polynomials are the outputs. These designs, in fact, produce identical output for any\(^1\)This shows that the product of numbers which are the sum of two squares is also a sum of two squares.
integer value of $x$. Can and how does one show equivalence of such
designs?

3.1. Polynomials and Polynomial Rings

The building blocks of polynomials are monomials:

**Definition** A monomial is a product of variables:

$$x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_n^{\alpha_n}$$

where $\alpha_i$ are non negative integers. This is abbreviated to $x^\alpha$ where $\alpha$ is the vector of exponents in the monomial.

**Definition** The total degree of a monomial is the sum of its exponents, denoted by $|\alpha|$.

$$x^\alpha = x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_n^{\alpha_n}$$

$$|\alpha| = \alpha_1 + \alpha_2 + \ldots + \alpha_n$$

**Definition** A polynomial is a finite sum of weighted monomials:

$$p = \sum_\alpha c_\alpha x^\alpha$$

where $c_\alpha$ belong to a set $k$. Examples of $k$ include the field of rational numbers, $\mathbb{Q}$, the field of real numbers, $\mathbb{R}$, the ring of integers, $\mathbb{Z}$ or the finite ring of integers modulo $p$ denoted $\mathbb{Z}_p$.

**Definition** The polynomial ring, denoted $k[x_1, x_2, \ldots, x_n]$, is the set of all polynomials in variables $x_1, x_2, \ldots, x_n$ with coefficients in a ring $k$. For example:

$$\frac{1}{2}x^2 + xy + \frac{1}{2}y^2 \in \mathbb{Q}[x, y]$$

**Definition** Degree, is defined for a polynomial $p$ in one variable (univariate), as the largest exponent power in $p$, denoted as $\text{deg}(p)$.

**Definition** Homogeneous polynomials — a polynomial is homogeneous if all its monomials have the same total degree. For example:

$$3x^3 - 4xy^2 + 5z^3$$
is a homogeneous polynomial of total degree 3.

**Definition** An ideal generated by a set of polynomials \{f_1, f_2, ..., f_s\}, where \( f_i \in k[x_1, x_2, ..., x_n] \), is denoted by \( \langle f_1, f_2, ..., f_s \rangle \) and is the span of the generating set:

\[
\langle f_1, f_2, ..., f_s \rangle = \{ p_1 f_1 + p_2 f_2 + ... + p_s f_s : p_1, p_2, ..., p_s \in k[x_1, x_2, ..., x_n] \}
\]

Ideals may have different generating sets but, in fact, may be identical. For example:

\[
\langle x - y^2, xy, y^2 \rangle = \langle x, y^2 \rangle
\]

This can be shown by demonstrating that every generating element in one ideal lives in the other:

\[
x - y^2 = (+1)x + (-1)y^2 \quad \rightarrow \quad x - y^2 \in \langle x, y^2 \rangle \\
x y = (y)x \quad \rightarrow \quad xy \in \langle x, y^2 \rangle \\
y^2 = y^2 \quad \rightarrow \quad y^2 \in \langle x, y^2 \rangle \\
x = (+1)(x - y^2) + (+1)y^2 \quad \rightarrow \quad x \in \langle x - y^2, xy, y^2 \rangle \\
y^2 = y^2 \quad \rightarrow \quad y^2 \in \langle x - y^2, xy, y^2 \rangle
\]

**Definition** Ideal Membership Testing — this is the task of ascertaining whether or not a given polynomial belongs to an ideal. For example:

\[
x^2 \in \langle x - y^2, xy \rangle \\
\text{due to} \quad x^2 = x(x - y^2) + y(xy)
\]

In general it can be difficult to determine whether two ideals are equal or perform ideal membership testing. Demonstrating that a polynomial belongs to an ideal requires dividing the polynomial of interest by one of the generating polynomials. To place this process on an algorithmic footing, a procedure for division is required where the polynomials are in more than one variable (multivariate).
3.2. Multivariate Division

Consider first division in the univariate case. Division of \( x^4 \) by \( x^2 - 3 \) can be performed by repeated subtraction of multiples of \( x^2 - 3 \):

\[
\begin{align*}
x^4 - x^2(x^2 - 3) &= 3x^2 \\
3x^2 - 3(x^2 - 3) &= 9
\end{align*}
\]

From which can be concluded that \( x^4 = (x^2 + 3)(x^2 - 3) + 9 \)

The result of the division is written as:

\[
x^4 = 9 \mod x^2 - 3
\]

In general, this division process will proceed as follows for \( f(x) \) divided by \( g(x) \):

\[
\begin{align*}
f - q_1g &= r_1 \\
r_1 - q_2g &= r_2 \\
&\vdots \\
r_{n-1} - q_ng &= r_n
\end{align*}
\]

From which can be concluded that \( f = (q_1 + q_2 + \ldots + q_n)g + r_n \)

where \( q_i \) is chosen such that \( \deg(r_i) \) decreases until \( \deg(r_i) < \deg(f) \). In order for \( r_i \) to decrease in degree, the polynomials \( q_i \) are chosen to cancel the leading terms in \( r_{i-1} \). Univariate division can be represented algorithmically
as follows:

\[
\begin{array}{l}
\textbf{Input } f, g \in k[x], g \neq 0 \\
\textbf{Output } q, r \text{ such that } f = qg + r, \deg(r) < \deg(g) \\
\text{begin} \\
q = 0, r = f \\
\text{while } r \neq 0 \text{ and } \deg(g) \leq \deg(r) \\
\text{begin} \\
q = q + \frac{\text{LT}(r)}{\text{LT}(g)} \\
r = r - \frac{\text{LT}(r)}{\text{LT}(g)} g \\
\text{end} \\
\text{end}
\end{array}
\]

\(\text{LT}(p)\) is the leading term of polynomial \(p\), namely the monomial of highest degree and its coefficient.

When attempting to modify this algorithm for multivariate division the question of defining the leading term of a multivariate polynomial arises. In order to do this an ordering of all monomials, called a \textit{term ordering}, is required, which states the conditions for \(x^\alpha < x^\beta\). The simplest term order is:

\textbf{Definition} Lexicographic Term Order (lex): This ordering proceeds by comparing powers of the variables in turn. Say \(x < y\), then comparing monomials will proceed by first comparing the power of \(y\) and then \(x\), hence:

\[1 < x < x^2 < x^3 < ... < y < xy < x^2y < ... < y^2 < ...
\]

In general if \(x_1 < x_2 < ... < x_n\) then

\[x_1^{\alpha_1}x_2^{\alpha_2}...x_n^{\alpha_n} < x_1^{\beta_1}x_2^{\beta_2}...x_n^{\beta_n} \iff \begin{cases} 
\text{there exists } j \text{ such that } \alpha_j < \beta_j \\
\text{and } \alpha_k = \beta_k \text{ for all } k > j
\end{cases}
\]

\textbf{Definitions} For a given term ordering and \(f \in k[x_1, x_2, ..., x_n], f \neq 0\) and
$0 \neq a_i \in k$ let:

$$f = a_1x^{\alpha_1} + a_2x^{\alpha_2} + ... + a_rx^{\alpha_r}$$

where $x^{\alpha_1} > x^{\alpha_2} > ... > x^{\alpha_r}$

- Leading Power Product $LP(f) = x^{\alpha_1}$
- Leading Coefficient $LC(f) = a_1$
- Leading Term $LT(f) = a_1x^{\alpha_1}$

Now, given a term ordering, the multivariate division algorithm for dividing $f$ by a set $F = \{f_1, f_2, ..., f_s\}$ can be described. If $f$ is unchanged by the division by $F$ then $f$ is said to be reduced with respect to $F$. The multivariate division algorithm is contained in Figure 3.1.

So $h$ is reduced by either canceling its leading term with multiples of the leading terms of $f_i$ or, if no reduction can occur, adding its leading term to the remainder $r$. Having performed the division $f$ has been reduced with respect to $F$, this is written as follows:

$$f \mod F = r$$

As an example consider dividing $f = x^2y^2$ by $f_1 = x^2y - x$ and $f_2 = xy^2 + y$ using lexicographic ordering $x > y$, the conclusion is:

$$f = yf_1 + xy$$

as $x^2y^2 = y(x^2y - x) + xy$

If the multivariate division had treated $f_1$ and $f_2$ in a reverse order the result would be:

$$f = xf_2 - xy$$

as $x^2y^2 = x(xy^2 + y) - xy$

This gives rise to the unfortunate conclusion that the remainder is dependent on the ordering of the $f_i$ polynomials. One could infer from the example
Input $f, f_1, f_2, \ldots, f_s \in k[x], f_i \neq 0$

Output $u_1, u_2, \ldots, u_s, r \in k[x]$ such that

$$f = u_1 f_1 + u_2 f_2 + \ldots + u_s f_s + r$$

$r$ is reduced with respect to $F$

begin

$u_1 = 0, u_2 = 0, \ldots, u_s = 0, r = 0, h = f$

while $h \neq 0$

begin

if there exists $i$ such that $LP(f_i)|LP(h)$ then

$i =$ least $i$ such that $LP(f_i)|LP(h)$

$u_i = u_i + \frac{LT(h)}{LT(f_i)}$

$h = h - \frac{LT(h)}{LT(f_i)} f_i$

else

$r = r + LT(h)$

$h = h - LT(h)$

endif

end

end

Figure 3.1.: Multivariate Division Algorithm.

that there does not exist polynomials $g_1$ and $g_2$ such that:

$$f = g_1 f_1 + g_2 f_2$$

as the division algorithm terminates without zero remainder. However, this is an incorrect conclusion:

$$f = \frac{1}{2} y f_1 + \frac{1}{2} x f_2$$

as

$$x^2 y^2 = \frac{1}{2} y (x^2 y - x) + \frac{1}{2} x (xy^2 + y)$$

In conclusion, the remainder of the multivariate division algorithm is dependent on the ordering of the polynomials $f_i$, even if a decomposition of
f with respect to $f_i$, does exist with zero remainder, the division algorithm may not return a remainder of zero. To address these problems and answer the motivating questions of this chapter we turn to Gröbner Bases.

### 3.3. Gröbner Bases

**Definition** For a set of polynomials $G = \{g_1, g_2, ..., g_s\}$ where $g_i \in k[x_1, x_2, ..., x_n]$ with a particular term order $G$ is a Gröbner Basis (GB) $\iff f \mod G$ is independent of the order of $g_i$ for all $f \in k[x_1, x_2, ..., x_n]$.

Using our previous example, the set $F = \{f_1 = x^2y - x, f_2 = xy^2 + y\}$ is not a Gröbner Basis with respect to the lexicographic term ordering as the remainder of $x^2y^2$ when divided by $F$ is not unique. Crucially it is observed that:

$$xy \in \langle x^2y - x, xy^2 + y \rangle \quad \text{however} \quad xy = xy \mod F$$

So $xy$ should have remainder zero when reduced modulo $F$ but will not be altered by the multivariate division algorithm. The cause is that while individually $f_1$ and $f_2$ cannot cancel $xy$, a linear combination of the two can. Such a combination is called an $S$-polynomial [16], and is defined as follows (where $LCM$ is the least common multiple):

$$S(f_1, f_2) = LCM(LT(f_1), LT(f_2)) \left( \frac{f_1}{LT(f_1)} - \frac{f_2}{LT(f_2)} \right)$$

$$= x^2y^2 \left( \frac{f_1}{xy^2} - \frac{f_2}{x^2y} \right)$$

$$= x(xy^2 + y) - y(x^2y - x)$$

$$= 2xy$$

By construction, these $S$-polynomials can not be reduced by $f_1$ or $f_2$ but yet reside in $\langle f_1, f_2 \rangle$. In fact this observation can be formalised into the following theorem:
Theorem 3.3.1  Buchberger’s Criterion [16]

For a set $G$ of polynomials defining an ideal $I \subset k[x_1, x_2, \ldots, x_n]$ with a particular term order

$$G \text{ is a Gröbner Basis} \iff S(g_i, g_j) \mod G = 0 \quad \text{for all} \quad g_i, g_j \in G$$

This gives rise to an algorithm which computes a Gröbner basis from any polynomial set, by repeatedly adding any $S$-polynomial which is non zero modulo $G$ into $G$. This algorithm is known as Buchberger’s algorithm and can be found in Figure 3.2.

<table>
<thead>
<tr>
<th>Input $F = {f_1, f_2, \ldots, f_s}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output Gröbner Basis $G$ such that $\langle G \rangle = \langle F \rangle$</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>$G = F$</td>
</tr>
<tr>
<td>while $H \neq G$ begin</td>
</tr>
<tr>
<td>$H = G$</td>
</tr>
<tr>
<td>for each pair $p, q \in H$</td>
</tr>
<tr>
<td>loop</td>
</tr>
<tr>
<td>$S = S(p, q) \mod H$</td>
</tr>
<tr>
<td>if $S \neq 0$ do</td>
</tr>
<tr>
<td>$G = G \cup {S}$</td>
</tr>
<tr>
<td>endloop</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

Figure 3.2.: Buchberger’s Algorithm.

This is the simplest form of Gröbner basis computation, many variants exist which improve on its computational complexity. Applying this algorithm to $\{x^2y - x, xy^2 + y\}$ using the lexicographic term ordering and
observing the set \( H \):

\[
\{ x^2 y - x, xy^2 + y \} \\
\{ x^2 y - x, xy^2 + y, 2xy \} \\
\{ x^2 y - x, xy^2 + y, 2xy, 2y \} \\
\{ x^2 y - x, xy^2 + y, 2xy, 2y, -2x \}
\]

The result is a Gröbner basis, however it can be further simplified to produce a reduced Gröbner basis:

**Definition** \( G = \{ g_1, g_2, \ldots, g_s \} \) is a Reduced Gröbner Basis if \( LC(g_i) = 1 \) and for all \( j \neq i \), \( LP(g_j) \) does not divide any term in \( g_i \).

It can be shown that for a given term order, every ideal has a unique reduced Gröbner basis. Thus reduced Gröbner bases provide a canonical representation of any ideal. The two questions regarding ideals can now be answered:

1. **Are two ideals identical?** Ideals are identical if and only if their reduced Gröbner bases are identical.

2. **How can Ideal Membership Testing be performed?** First compute the Gröbner Basis \( G \) for the ideal \( I \), then compute \( r = f \mod G \).

   Now \( f \in I \) if and only if \( r = 0 \), as the result of reduction modulo \( G \) is unique.

For example, it was shown earlier that \( \langle x - y^2, xy, y^2 \rangle = \langle x, y^2 \rangle \), this can be confirmed by computing a reduced Gröbner basis for \( \{ x - y^2, xy, y^2 \} \), the result is \( \{ x, y^2 \} \). Performing such calculations using the computer algebra package Singular [43]:

```plaintext
ring r=0,(x,y,z),lp; // 0,(x,y,z) refers to the ring Q[x,y,z]
// lp refers to the lex term ordering
ideal I = x-y^2,xy,y^2;
groebner(I);
```

which returns:

```
_[1]=y^2
_[2]=x
```

The runtimes, degrees of the resultant polynomials and their number when computing Gröbner bases is highly dependent on the term order.
3.4. Elimination and Solutions to Systems of Multivariate Polynomials

Returning to our motivating Q1: how can decompositions be discovered, such as:

Let: \[ u = a^2 + b^2 \]
\[ v = c^2 + d^2 \]

Then: \((ac - bd)^2 + (ad + bc)^2 = uv\)

Consider the following expansion:

\[
(a^2 + b^2) (c^2 + d^2) = (u - a^2 - b^2 - u) (v - c^2 - d^2 - v)
= (u - a^2 - b^2) (v - c^2 - d^2)
- v (u - a^2 - b^2) - u (v - c^2 - d^2) + uv
\]

Thus: \((a^2 + b^2) (c^2 + d^2) = uv \mod \{u - a^2 - b^2, v - c^2 - d^2\}\)

However, this reduction modulo \(\{u - a^2 - b^2, v - c^2 - d^2\}\) will only return \(uv\) if the term order weights \(u\) and \(v\) lower than \(a, b, c\) or \(d\). This requirement gives rise to the need for the following definition:

**Definition** An Elimination Order which places \(x\) variables larger than \(y\) variables satisfies, for \(X_1, X_2\) monomials in \(x\) variables and \(Y_1, Y_2\) monomials in \(y\) variables:

\[
X_1 Y_1 < X_2 Y_2 \iff \begin{cases} 
X_1 <_x X_2 \\
\text{or} \\
X_1 = X_2 \text{ and } Y_1 <_y Y_2
\end{cases}
\]

where \(<_x\) and \(<_y\) are term orders on variables \(x\) and \(y\).

An answer to our motivating Q1 can now be given, **How and can polynomial \(f\) be expressed as a polynomial in \(p_1, p_2,\ldots, p_s\)?** Letting \(f, p_i \in k[x_1, x_2, \ldots, x_n]\):

1. Compute a Gröbner basis, \(G\) for \(F = \langle y_1 - p_1, y_2 - p_2, \ldots, y_s - p_s \rangle\) using an Elimination Order which places \(x\) variables larger than \(y\) variables.
2. Compute \(r = f \mod G\).
3. If $f$ can be expressed only in terms of $p_i$ then $r$ will only contain $y$ variables.

4. Replacing $y_i$ by $p_i$ in $r$ provides the decomposition.

For example consider:

$$f = \left(\frac{a + b}{2}\right)^2$$

$$p_1 = \frac{a - b}{2} \quad p_2 = ab$$

$$F = \{x - \frac{a - b}{2}, y - ab\}$$

$$G = \{a - b - 2x, b^2 + bx - y\} \quad \text{lex} \quad a > b > x > y$$

$$f \mod G = x^2 + y$$

So $f$ can be expressed in terms of $p_1$ and $p_2$ with decomposition:

$$\left(\frac{a + b}{2}\right)^2 = \left(\frac{a - b}{2}\right)^2 + (ab)$$

In effect this technique is a change of basis. This transformation can be achieved in Singular as follows:

```singular
ring r=0,(a,b,x,y),lp;
poly f = (1/4)*(a+b)^2;
ideal I = x-(1/2)*(a-b),y-ab;
ideal J = groebner(I);
reduce(f,J);
> x^2+y
```

Elimination orders have a crucial property which will allow us to solve systems of multivariate polynomial equations. First note that if $G = \{g_1, g_2, ..., g_r\}$ is a Gröbner basis for $F = \{f_1, f_2, ..., f_s\}$ then, by design and because they share the same reduced Gröbner basis:

$$\langle F \rangle = \langle G \rangle \quad\implies\quad f_i \in \langle G \rangle \quad g_i \in \langle F \rangle \quad\implies\quad f_1 = f_2 = ... = f_s = 0 \iff g_1 = g_2 = ... = g_r = 0$$
It can be concluded that solving the system of equations defined by $f_i = 0$ is identical to solving the set $g_i = 0$. Consider the following example:

$$ F = \{ x^2 + y^2 + z^2 - 1, z - x^2 y^2, x + y + z \} $$

$$ G = \{ x + y + z, 2y^2 + 2yz + 2z^2 - 1, 4z^4 - 4z^2 - 4z + 1 \} \quad \text{lex} \quad x > y > z $$

So if it is required to solve $f_i = 0$, then one can solve $g_i = 0$. For this Gröbner basis this process is straight forward.

$$ 4z^4 - 4z^2 - 4z + 1 = 0 \quad 2y^2 + 2yz + 2z^2 - 1 = 0 \quad x + y + z = 0 $$

These can be solved in turn, solving the quartic in $z$ will produce four solutions $z_1, z_2, z_3$ and $z_4$. For each of these solutions, substitute into the second equation. Solving the quadratic in $y$ will provide two values for $y$. Finally $x$ will satisfy $x = -y - z$. Numerically, to three decimal places, four solutions are (swapping $x$ and $y$ produces another four):

$$ x = -0.393 \pm 0.766i \quad y = 1.135 \mp 0.142i \quad z = -0.743 \mp 0.623i $$

$$ x = -0.638 + 0.850i \quad y = -0.638 - 0.850i \quad z = 1.277 $$

$$ x = 0.579 \quad y = -0.788 \quad z = 0.208 $$

Notice how the Gröbner basis was triangular in the sense that one of the terms only involved $z$, another only $z$ and $y$ and the last all three variables. This is all a result of the term order. This property is proven by the following theorem:

**Theorem 3.4.1** Let $G$ be a Gröbner basis for an ideal $I$ of $k[y_1, ..., y_m, x_1, ..., x_n]$ with respect to an elimination order placing $x$ variables larger than $y$ variables then

$$ G \cap k[y_1, ..., y_m] \quad \text{is a Gröbner Basis for the ideal} \quad I \cap k[y_1, ..., y_m] $$

This is a subtle yet important property of elimination orders. If there are polynomials in $J$ which are only a function of variables $y_i$, then a Gröbner basis for such polynomials are all polynomials in $G$ which only contain $y$ variables. So if there are a finite number of solutions to $f_i = 0$ then computing $G$ with respect to a lexicographic ordering will produce a set of polynomials, one of which will only be a function of one variable, another
of two, another of three and so on. Thus the triangular property observed is an essential feature of Gröbner Basis calculation with lex ordering. Note that if there are an infinite set of solutions to \( f_i = 0 \) then the \( g_i \)'s will still be triangular but a univariate polynomial will not be among them. For example:

\[
F = \{ x^2 + y^2 + z^2 - 2x - 2y - 2z + 1, \\
x^2 + y^2 + z^2 - 6x - 6y - 6z + 25, x + y + z - 6 \}
\]

\[
G = \{ x + y + z - 6, 2y^2 + 2yz + 2z^2 - 12y - 12z + 25 \} \text{ lex } x > y > z
\]

In this case, the second polynomial in \( G \) can be solved for \( y \) in terms of \( z \) and finally \( x = 6 - y - z \). If there are no solutions to \( f_i = 0 \) then the Gröbner basis calculation will return \( G = \{1\} \), which clearly gives rise to a contradiction if it is attempted to solve \( g_i = 0 \).

### 3.5. Membership of Vanishing Ideals

Our motivating Q2 concerns the equivalence of polynomials over a finite ring:

\[
y_1[2 : 0] = x^4 + 2x \\
y_2[2 : 0] = 2x^3 + x^2
\]

Equivalence of such polynomials amounts to their difference vanishing over \( \mathbb{Z}_2^3 \):

\[
y_1 - y_2 = x^4 - 2x^3 - x^2 + 2x = 0 \pmod{2^3} \\
x^4 - 2x^3 - x^2 + 2x = 0 \text{ over the ring } \mathbb{Z}_2^3[x]
\]

The vanishing ideal is the set of all such polynomials:

**Definition** Vanishing Ideal, \( I_c \):

\[
I_{\text{vanish}} = \{ q \in \mathbb{Z}_p[x_1, x_2, \ldots, x_n] : q(x_i) = 0 \pmod{\mathbb{Z}_p} \forall x_i \in \mathbb{Z}_p \}
\]

Formal verification of circuits, \( y_1 \) and \( y_2 \) that are expressible as polynomials in \( \mathbb{Z}_p[x_1, x_2, \ldots, x_n] \) reduces to performing ideal membership testing for this vanishing ideal. Members of the one dimensional vanishing ideal
can be constructed as follows (note that \((n)_r\) represents the falling factorial \((n)_r = n(n-1)...(n-r+1)\):

\[
\binom{n}{r} \in \mathbb{Z} \\
\implies \frac{(n)_r}{r!} \in \mathbb{Z} \\
\implies r! \mid (n)_r \\
\implies (n)_r = 0 \mod \mathbb{Z}_p \text{ for all } n \in \mathbb{Z} \text{ if } p|r!
\]

As this property holds for every integer \(n\) then the following polynomial will also vanish whenever \(x \in \mathbb{Z}\):

\[x(x-1)...(x-r+1) = 0 \mod \mathbb{Z}_p \text{ for } x \in \mathbb{Z} \text{ if } p|r!\]

The notation \((x)_r\) is used for such polynomials. At this point, it is useful to define the Smarandache Function:

**Definition** Smarandache Function -

\[SF(p) = \min\{n : p|n!\}\]

It can then be concluded that:

\[(x)_{SF(p)} \in I_p^{\text{vanish}} \]

\[\text{e.g. } x(x-1)(x-2)(x-3) = 0 \mod \mathbb{Z}_3\]

From this point forward it is assumed that \(p = 2^k\). In this particular case the calculation of \(SF(p)\) can be simplified by using a helper function which returns the number of times two divides a given number \(#_2(k) = \max\{n : 2^n|k\}\):

\[SF(2^k) = \min\{n : 2^k|n!\} = \min\{n : #_2(n!) \geq k\}\]

Now the number of times 2 divides \(n!\) can be calculated by first counting the number of numbers \(\leq n\) which are divisible by 2, adding this to the number of numbers \(\leq n\) which are divisible by 4, adding this to the number
of numbers \( \leq n \) which are divisible by 8 \textit{etc.} Hence:

\[
SF(2^k) = \min \left\{ n : \sum_{i=1}^{\text{length} \, j} \left\lfloor \frac{n}{2^i} \right\rfloor \geq k \right\}
\]

\[
= \min \{ n : (n >> 1) + (n >> 2) + (n >> 3) + \ldots \geq k \}
\]

where \( >> \) is a right shift on the binary representation of \( n \). Letting \( n = n_jn_{j-1}n_1n_0 \) be the binary expansion of \( n \) then:

\[
SF(2^k) = \min \left\{ n : n_jn_{j-1}n_1n_0 \geq k \right\}
\]

\[
= \min \left\{ n : \sum_{i=1}^{j} n_i (2^i - 1) \geq k \right\}
\]

\[
= \min \left\{ n : \left( \sum_{i=1}^{j} n_i 2^i \right) - \left( \sum_{i=1}^{j} n_i \right) \geq k \right\}
\]

\[
= \min \{ n : n - \text{Hamm}(n) \geq k \}
\]

(3.1)

where \( \text{Hamm}(n) \) is the Hamming weight of \( n \). Note that \( SF(2^k) \) is of the order of \( k \) which is an exponentially smaller. Further, in the very common case, when working modulo \( 2^{2k} \) this can be further evaluated as:

\[
SF \left( 2^{2k} \right) = \min \left\{ n : n - \text{Hamm}(n) \geq 2^k \right\} = 2^k + 2
\]

(3.2)

In such cases it can be concluded that:

\[
x(x-1)(x-2)(x-9) = 0 \mod \mathbb{Z}_{2^8}
\]

\[
x(x-1)(x-2)(x-17) = 0 \mod \mathbb{Z}_{2^{16}}
\]

\[
x(x-1)(x-2)(x-33) = 0 \mod \mathbb{Z}_{2^{32}}
\]

\[
\ldots
\]

\[
(x)_{2^{k+2}} = 0 \mod \mathbb{Z}_{2^k}
\]

Lower degree polynomials can also be constructed that also belong to the vanishing ideal by multiplying by appropriate powers of two. Consider \( (x)_3 \), this is divisible by \( 2^{\#x(3)} = 2 \) for any integer \( x \), therefore \( 4(x)_3 \) is divisible.
by $2^3$ and hence $4(x)_{3} \in I_{2^3}^{\text{vanish}}$. Similarly the following set of polynomials can be shown to lie within $I_{2^3}^{\text{vanish}}$:

$$\{(x)_{4}, 4(x)_{3}, 4(x)_{2}, 8x, 8\} \in I_{2^3}^{\text{vanish}}$$

In general the following set of polynomials belong to $I_{2^k}^{\text{vanish}}$:

$$H_{k} = \left\{ h_{n} = 2^{k-n+\text{Hamm}(n)}(x)_{n} : n = 0, 1, 2, ..., SF(2^k) \right\}$$

$H_{k}$ is actually a basis for $I_{2^k}^{\text{vanish}}$, in order to demonstrate this first note that the $(x)_{n}$ form a polynomial basis:

**Theorem 3.5.1** [69] Every one dimensional polynomial with integer coefficients can be written as a linear combination of falling factorials in $x$:

$$p \in \mathbb{Z}[x], \deg(p) = n \implies \exists a_{i} \in \mathbb{Z} \text{ such that } p = \sum_{i=0}^{n} a_{i}(x)_{i}$$

**Proof** By induction, the theorem trivially holds for $n = 0$. Assuming it holds for all integer values up to $n$ then considering polynomial $p$ of degree $n + 1$:

$$p = a_{n+1}x^{n+1} + g \text{ for some } a_{n+1} \in \mathbb{Z} \text{ and polynomial } g \text{ of degree } n$$

$$= a_{n+1}(x)_{n+1} + a_{n+1} \left( x^{n+1} - (x)_{n+1} \right) + g$$

noting that $x^{n+1} - (x)_{n+1}$ is a polynomial of degree $n$ then:

$$= a_{n+1}(x)_{n+1} + \sum_{i=0}^{n} b_{i}(x)_{i} \text{ for some } b_{i} \in \mathbb{Z} \quad \square$$

In fact, in can be shown that $H_{k}$ is actually a Gröbner basis over $\mathbb{Z}_{2^k}[x]$.
As an example note that $H_3$ is a basis for $I_{2^3}^{\text{vanish}}$:

$$p \in I_{2^3}^{\text{vanish}}$$

$$\implies p(x) = \sum_{i=0}^{n} a_i(x)_i \equiv 0 \pmod{8} \text{ for some } a_i \in \mathbb{Z}$$

$$\implies p(0) = a_0 \equiv 0 \pmod{8}$$

$$\implies p(1) = a_1 + a_0 = a_1 \equiv 0 \pmod{8}$$

$$\implies p(2) = 2a_2 + 2a_1 + a_0 = 2a_2 \equiv 0 \pmod{8}$$

$$\implies p(3) = 6a_3 + 6a_2 + 3a_1 + a_0 = 6a_3 \equiv 0 \pmod{8}$$

From which can be concluded that for some $a_i \in \mathbb{Z}$:

$$p = 8b_0 + 8b_1 x + 4b_2(x)_2 + 4b_3(x)_3 + \sum_{i=4}^{n} a_i(x)_i$$

$$\implies p \in \langle 8, 8x, 4(x)_2, 4(x)_3, (x)_4 \rangle = H_3$$

In the general case:

$$p \in I_{2^k}^{\text{vanish}}$$

$$\implies p(x) = \sum_{i=0}^{n} a_i(x)_i \equiv 0 \pmod{2^k} \text{ for some } a_i \in \mathbb{Z}$$

$$\implies p(r) = \sum_{i=0}^{n} a_i(r)_i \equiv 0 \pmod{2^k} \text{ for } r = 0, 1, 2, ..., SF(2^k) - 1$$

$$\implies \sum_{i=0}^{r} i!a_i \left(\begin{array}{c} r \\ i \end{array}\right) \equiv 0 \pmod{2^k} \text{ for } r = 0, 1, 2, ..., SF(2^k) - 1$$

$$\implies 2^{k\lceil r!a_r \rceil} \equiv 0, 1, 2, ..., SF(2^k) - 1$$

$$\implies \#2(r!a_r) \geq k \text{ for } r = 0, 1, 2, ..., SF(2^k) - 1$$

$$\implies \#2(a_r) \geq k - r + \text{Hamm}(r) \text{ for } r = 0, 1, 2, ..., SF(2^k) - 1$$

From which can be concluded that for some $b_i \in \mathbb{Z}$:

$$p = \sum_{i=0}^{SF(2^k)-1} b_i 2^{k-i+\text{Hamm}(i)}(x)_i + \sum_{i=SF(2^k)}^{n} a_i(x)_i$$

$$\implies p \in \langle 2^{k-i+\text{Hamm}(i)}(x)_i : i = 0, 1, 2, ..., SF(2^k) \rangle = H_k$$

This argument proves that not only is $H_k$ a Gröbner basis for the vanishing
ideal over $\mathbb{Z}_{2^k}[x]$ but further:

$$p(x) = 0 \mod 2^k \iff p(x) = 0 \mod 2^k$$
for all $x \in \mathbb{Z}$

for $r = 0, 1, 2, ..., SF(2^k) - 1$

Analogously, a Gröbner basis for the vanishing ideal for $\mathbb{Z}_{2^k}[x_1, x_2, ..., x_n]$ can be computed. In non reduced form, it is:

$$\{2^r(x_1)^{\alpha_1}(x_2)^{\alpha_2}...(x_n)^{\alpha_n} : 2^k | 2^r \alpha_1! \alpha_2! ... \alpha_n!\}$$

and similarly it can be shown that:

$$p(x) = 0 \mod 2^k \iff p(x) = 0 \mod 2^k$$
for all $x \in \mathbb{Z}^n$

for $x \in [0, 1, 2, ..., SF(2^k) - 1]^n$ (3.3)

This concludes the preliminaries chapter. The method for solving systems of multivariate polynomials presented in this chapter is used throughout the thesis along with the elimination techniques. The results on equivalence of polynomials in the ring $\mathbb{Z}_{2^k}$ play a crucial role in a contribution to formal verification presented in the next chapter.
4. Lossless Fixed-Point Polynomial Optimisation

This chapter concerns the lossless implementation of polynomials with integer coefficients. Some of the most prolific datapath operations can be expressed as fixed-point polynomials. These include adders, subtractors, multipliers, squarers, multiply-accumulators (MACs), chained additions, decrementors, incrementors, fixed-point interpolation, etc. Fixed-point polynomials also occur within floating point modules, e.g. multiply accumulate, dot product; see [87] for an example. These polynomials involve sign magnitude quantities \((-1)^a\) and can be viewed as requiring the implementation of a set of mutually exclusive polynomials, e.g. \(s? - ab : ab\). Arithmetic Logic Units (ALUs) implement a set of fundamental operations mutually exclusively. Hardware supporting multiple standards or reuse for a variety of algorithms may require the implementation of mutually exclusive operations. For these reasons, our design of interest is, for some polynomials \(p_i\) in variables \(x_1, x_2... x_n\) with integer coefficients and a select signal \(s\):

\[
y = \begin{cases} 
  p_0(x_1, x_2, ..., x_n) & \text{if } s == 0 \\
  p_1(x_1, x_2, ..., x_n) & \text{if } s == 1 \\
  \vdots & \\
  p_{m-2}(x_1, x_2, ..., x_n) & \text{if } s == m - 2 \\
  p_{m-1}(x_1, x_2, ..., x_n) & \text{if } s == m - 1
\end{cases}
\]  

(4.1)

where it is assumed that \(x_i\) are unsigned and of equal bit width (the optimisation procedure described in this chapter can be modified to remove these restrictions, however the exposition of such modifications is beyond the scope of this chapter).

In regard to this implementation challenge the literature is split between polynomial manipulations and datapath operator design. In terms of operat-
tor design, a fundamental building block of fixed-point polynomial datapath is the arithmetic sum-of-products (SOP). An early example of how an SOP may be implemented by summing all partial product bits in parallel can be found in the integer part of a floating point multiply accumulator [3]. Previous work has considered improvements to the final carry propagate adder of an SOP [38]. In [36], inverted partial product arrays were shown to improve quality of results. Designs implementing operations of the form $\sum k_i x_i y_i$ where $k_i$ are constants and $x_i$ and $y_i$ are input operands have been considered in [104]. In [104], multiplication by a constant is performed by using the canonical signed digit recoding and $x_i y_i$ is computed in redundant carry-save form. There is a wealth of design options for SOP or POS (product-of-sum) expressions by manipulating the Booth encoded multipliers in a variety of styles [181].

Despite the existence of efficient implementations of SOP and POS expressions, most datapath synthesis techniques cannot exploit these highly optimised blocks due to non SOP expressions found within the datapath, e.g. muxing and shifting. In [174], data flow graphs have been locally manipulated to increase the proportion of the datapath which can be expressed as a single SOP, hence reducing delay and area. For example one of the transformations includes $(a + b + c) << d = (a << d) + (b << d) + (c << d)$, hence shifters can be moved through summations, a fact exploited fully in [35].

In terms of considering mutually exclusive SOP expressions, an example can be found in [174]: $\text{sel}\?a + b : c = (\text{sel}\?a : c) + (\text{sel}\?b : 0)$. However such optimizations were restricted to localized regions. A fuller consideration of merging mutually exclusive operations can be found in [37]. In [37], the SOP is split into partial generation, array reduction and final carry propagate adder with muxing on inputs to each of these units.

Implementing polynomials by algebraic manipulation has had considerable interest. Common sub-expression extraction techniques have been considered in [79] and attention has been given to exploiting the finite ring in which the polynomial resides [67]. A canonical description of polynomials over the integer ring has been developed, termed modular Horner expansion diagrams. These provide a platform in which to perform optimisation and verification of fixed point polynomials, [7], [143] and [76]. An integer linear program has been used to find suitable linear building blocks from which to build polynomials over the integer ring in [68]. Gröbner bases have been
used to perform component matching and use [138].

The contributions of this chapter are orthogonal to the majority of this previous research and can be viewed as an RTL to RTL transformation which would fit into an existing synthesis flow. To reduce the implementation cost of the design stated in equation (4.1), the aim is to recast this into a form where inputs are selected between and then a single polynomial is implemented (here $x_{i,j}$ are drawn from $x_1, x_2, ..., x_n$):

$$
\tilde{x}_1 = (s == 0)? x_{1,1} : (s == 1)? x_{2,1} : \ldots : x_{m,1} \\
\tilde{x}_2 = (s == 0)? x_{1,2} : (s == 1)? x_{2,2} : \ldots : x_{m,2} \\
\ldots \\
\tilde{x}_r = (s == 0)? x_{1,r} : (s == 1)? x_{2,r} : \ldots : x_{m,r} \\
y = p(\tilde{x}_1, \tilde{x}_2, ..., \tilde{x}_r) \tag{4.2}
$$

where $p$ is a polynomial which is then amenable to all the manipulations previously mentioned.

In this chapter the implementation and properties of a SOP are explored to motivate the optimisation that follows. This is then followed by a motivating example in Section 4.2, the elements of which are then expanded and generalised in subsequent sections. These sections deal with how the $\tilde{x}$ variables are optimally constructed in Section 4.3, how optional negations are handled in Section 4.4 and a statement of the overall flow in Section 4.5. Having established the optimisation, attention is then turned to how the transformations can be formally verified in Section 4.6. The chapter closes with experimental evidence on the benefits of the proposed optimisation and verification technique in Section 4.7.
4.1. Arithmetic Sum-Of-Products

The fundamental building block of datapath designs, the integer arithmetic sum-of-products, is of the form:

\[ \sum_i (-1)^{s_i} 2^{r_i} x_i y_i \]

for constants \( s_i \in \{0, 1\} \) and integers \( r_i \geq 0 \) and \( x_i \) and \( y_i \) which are signed or unsigned variables. The implementation of the expression begins with array creation, an example array can be found in Figure 4.1 which is the array for the following SOP:


Figure 4.1.: Example Sum-Of-Products Array.

where each dot represents an AND gate which produces \( x_i[j] \wedge y_i[k] \). Each row is termed a partial product. Once the array is formed, array reduction is performed by the use of reduction cells such as a Full Adder (FA) which produces the sum of three bits and Half Adder (HA) which produce the sum of two bits. Repeated use of reduction cells reduces the array to a height of two, as in Figure 4.2.
After array reduction, an integer adder is used to sum the array of height two.

From an appreciation of the structure of the arithmetic SOP particular properties can be inferred. To provide evidence for these inferences a set of designs were synthesised by Synopsys Design Compiler 2012.06-SP2, [162], in ultra mode using the TSMC 40nm library Tcnn40lpbpw. The results can be found in Table 4.1 where $a$, $b$, $c$ and $d$ are unsigned 16 bit inputs, $x$ and $y$ are unsigned 32 bit inputs and $s$ is a 1 bit input. From the SOP implementation exposition the following conclusions can be drawn:

- **Significant hardware benefits arise from the efficient SOP implementation.** If an SOP is implemented by producing the prod-
Table 4.1.: Synthesis Results for Sample SOPs.

<table>
<thead>
<tr>
<th>Design</th>
<th>Delay (ns)</th>
<th>Area (µm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.70</td>
<td>2062</td>
</tr>
<tr>
<td>2</td>
<td>0.62</td>
<td>533</td>
</tr>
<tr>
<td>3</td>
<td>1.76</td>
<td>2171</td>
</tr>
<tr>
<td>4</td>
<td>1.77</td>
<td>2092</td>
</tr>
<tr>
<td>5</td>
<td>1.75</td>
<td>2176</td>
</tr>
<tr>
<td>6</td>
<td>2.05</td>
<td>4062</td>
</tr>
<tr>
<td>7</td>
<td>2.14</td>
<td>4667</td>
</tr>
<tr>
<td>8</td>
<td>1.94</td>
<td>2562</td>
</tr>
<tr>
<td>9</td>
<td>2.05</td>
<td>2513</td>
</tr>
<tr>
<td>10</td>
<td>1.94</td>
<td>2398</td>
</tr>
</tbody>
</table>

- Introducing extra bits into a SOP will produce negligible effects on the implementation cost. The SOP implementation uses a single array where all partial product bits are summed in parallel, thus the introduction of a number of bits which is small with respect to the total number of partial product bits will negligibly affect implementation cost. For example, comparing Designs 3 and 4 to Design 1 shows that adding one extra variable incurs negligible delay and area degradation.

- Internal products do not exist. Given the SOP implementation creates and reduces a single array, any internal products will not exist during the reduction. A design which returns products as well as their sum will have higher implementation cost; Design 7 requires 15% more area than Design 6. Adding the result of a selected product into a sum will incur significant delay costs, comparing Design 8 to 3 shows a 10% delay and 18% area degradation.

- Optional negation occurring in the middle of an SOP should
be avoided. The SOP implementation contains a single array reduction, modifying signals that would be the result of a reduction of part of the array will incur the cost of a full binary addition. Thus optional negation occurring in the middle of an SOP should be avoided. By writing \( s? - ab : ab = (s? - a : a)b = (a \oplus s + s)b \) which effectively moves an internal optional negation to an input negation, Designs 10 and 9 show this saves 5% delay and area.

- **XORing the result of a SOP has negligible cost** The final gate on the critical path of a two input adder is an XOR, say \( A \oplus B \) where \( A \) is critical and \( B \) is non critical. Observing that \((A \oplus B) \oplus s = A \oplus (B \oplus s)\), then XORing a result of a SOP which terminates in a two input adder becomes non critical. Note the negligible difference Designs 4 and 5.

4.2. Motivational Example

In order to motivate the subsequent algorithm, consider the following simple mutually exclusive set of SOPs:

\[
m_1 = (s == 0)? \; ab + c :
\]
\[
(s == 1)? \; bc - a :
\]
\[
(s == 2)? \; c - ab :
\]
\[
- a - bc
\]

Given the knowledge from the previous section on the properties of SOP implementations, can these equations be reformulated in such a way that the design’s hardware resource utilisation is as close to that of \( ab + c \) as possible? That is, it is desired to minimise the cost of any muxing and negation that is introduced. To this end, the equations will be reformulated such that only one SOP is required. The first step in doing this, is to reorder the multiplications and additions in such a way to minimise the amount of muxing between operands once merged. Consider the following rewrite, where each mutually exclusive polynomial has been written in the form \( AB + C \) and choosing the order of \( A \) and \( B \) such that the second term
in the multiplication is always $b$.

$$m_2 = \begin{cases} 
  ab + c & (s == 0) \\
  cb - a & (s == 1) \\
  -ab + c & (s == 2)
\end{cases}$$

Having rewritten the polynomials in this way allows them to be merged together into a single SOP by using the standard two’s complement identity $-x = \overline{x} + 1$ then $(-1)^{neg}a = a \oplus neg + neg$. Note also that $s$ is assumed to be two bits in length and $s_1$ and $s_0$ denotes the most and least significant bits of $s$ respectively. This leads to the following reformulation:

$$A = s_0?c : a$$
$$C = s_0?a : c$$

$$m_3 = (-1)^{s_1} A b + (-1)^{s_0} C$$
$$= A(b \oplus s_1) + A s_1 + (C \oplus s_0) + s_0$$

This reformulation requires the optional negation of the product, but the product is the most delay and area expensive part of the SOP. To minimise this cost it is required to minimise the logic that provides inputs to the product; hence it would be advantageous to rewrite the SOPs such that the product is always positive. This can be achieved by using another negation identity; consider replacing $x$ with $x - 1$ in the formula $-x = \overline{x} + 1$, simplifying gives rise to $-x = \overline{x - 1}$. This freedom can be exploited to rewrite each mutually exclusive polynomial such that the product is always positive as follows:

$$ab + c$$
$$cb - a = cb + \overline{a} + 1$$
$$-ab + c = -(ab + c + 1) = \overline{ab + c}$$
$$-cb - a = -(cb + a) = \overline{cb + a - 1}$$
Substituting these back into the original problem statement:

\[
m_4 = (s == 0)? ab + c : \\
   (s == 1)? cb + a + 1 : \\
   (s == 2)? \overline{ab + \overline{c}} : \\
   \overline{cb + a - 1}
\]

Merging these polynomials together produces the following formulation:

\[
A = s_0?c : a \\
C = (s_0?a : c) \oplus (s_1 \oplus s_0) \\
m_5 = (Ab + C + s_1 \oplus s_0 - s_1) \oplus s_1
\]

Note how in this formulation the operations on the product have been reduced, these have been replaced by modifications to the \( C \) variable. The results from the previous section show that the final XOR and the introduction of the bits \( s_1 \oplus s_0 \) and \( s_1 \) into the partial product array will have negligible impact. It is expected that the reformulation \( m_5 \) will have an implementation cost close to that of \( AB + C \).

A sample logic synthesis of \( m_1, m_5 \) and \( ab + c \) using a leading synthesis tool shows that \( m_1 \) has a hardware area implementation cost four times larger than that of \( ab + c \), whereas \( m_5 \) is only 28% larger than \( ab + c \). More details on the synthesis results for this example are presented in the experiments section of this chapter. Given the benefits of this particular optimisation, it is natural to explore the generalisation of this approach in the hope that similar benefits can be attained in a general case. The techniques used for the motivating example are now generalised in the following sections. In this motivating example, each mutually exclusive polynomial had the same number of terms and total degrees, in the next section the general case of a set of arbitrary polynomials is considered and how to reorder them prior to merging. This is followed by a section on dealing with optional negations in general, after which the overall flow is presented.
4.3. Control Logic Minimisation

Minimising the cost of selection is not only desirable for minimising selection hardware but also enabling polynomial rewriting techniques. For example:

\[
y = s \cdot (s \? a : d)(b + c)
\]

Now \(y_1\) and \(y_2\) implement the same polynomial, however \(y_1\) implements a polynomial of the form \(A(B + C)\) where \(y_2\) requires implementing the potentially more expensive polynomial \(AB + CD\). Minimising input selection maximises the exposed correlations between the inputs to the polynomial \(p\) and hence minimises the potential implementation cost. Initially the following assumptions are made, as this simplifies the exposition.

- All the polynomials \(p_i\) have the same number of terms.
- All coefficients are one.
- No two monomials are equal.

These restrictions are lifted in the next section. The selection hardware may be reduced at the expense of the polynomial implementation cost:

\[
y = (s == 0)\? a + b : (s == 1)\? b + c : c + a
\]

Now \(y_1\) and \(y_2\) implement the same polynomial, however \(y_2\) has simpler selection logic (selecting between a non zero element and zero) than \(y_1\). However, \(y_2\) requires a three input sum versus \(y_1\) which only requires a two input sum. In the general case, the number of terms in \(p\) can grow such that the \(p_i\) occupy disjoint parts of \(p\), for example, consider \(y_3\):

\[
y_3 = ((s == 0)?a : 0) + ((s == 0)?b : 0) +
     ((s == 1)?b : 0) + ((s == 1)?c : 0) +
     ((s == 2)?c : 0) + ((s == 2)?a : 0)
\]
where each addend is non-zero for only one value of $s$. The selection logic has been significantly reduced, but $p$ is considerably more complex than $p_i$. Thus the selection logic may be reduced at the expense of the implementation cost of the polynomial. Typically, the growth in $p$ outweighs the benefit of selection hardware reduction. It is thus desired to keep the number of terms of $p$ equal to the maximum number of terms of $p_i$. A compact algebraic formulation of this property is $p(\mathbf{1}) = \max_i p_i(\mathbf{1})$ where $\mathbf{1}$ is a vector where all entries are one. With this constraint, the optimisation that precisely embodies the problem can be stated as:

$$
\text{Given } p_i, \text{ select } x_{j,i} \text{ and } p \text{ minimising }
\sum_i \text{num of distinct elements in } \{x_{1,i}, x_{2,i}, \ldots, x_{m,i}\}
$$

subject to $p(\tilde{x}(s, \mathbf{x})) = y(s, \mathbf{x})$ for all $s$ and $\mathbf{x}$

and $p(\mathbf{1}) = \max_i p_i(\mathbf{1})$ (4.3)

4.3.1. Determining the Total Degrees of Monomials in $p$

The number of terms and total degrees of the monomials in $p$ must be such that a particular assignment to the inputs of $p$ results in returning $p_i$ for all $i$. This places a restriction on the number of terms and total degrees of the monomials in $p$, however there are many such $p$. Now $p$ should be chosen to be just large enough to be able to express the same value of each of the polynomials $p_i$, otherwise the implementation costs of $p$ will be unnecessarily large. Through some examples, an algorithm is now constructed which establishes the number of terms and total degrees of the monomials in $p$, whose number of terms does not exceed that of all the polynomials $p_i$. Let $d_i$ be the multiset of total degrees of monomials in $p_i$ and $d$ be the multiset of total degrees of monomials in $p$. Note that as the total degrees of monomials may have repeated elements these are multisets, where repeated elements may be present. To motivate the algorithmic construction of $d$, consider examples of $d_i$. Firstly, where $p_i$ are all monomials, so for some
non negative integers $t_i$:

\[ d_0 = \{3\} \quad d_1 = \{2\} \quad d_2 = \{4\} \implies d = \{4\} \quad \text{// maximum of the total degrees} \]

e.g. \[ y = (s == 0)? \quad abc : (s == 1)? \quad bc : \quad abcd \]
\[ = bc((s == 1)?a)((s > 1)?d : 1) \]
\[ d_0 = \{t_0\} \quad d_1 = \{t_1\} \quad \ldots \quad d_{m-1} = \{t_{m-1}\} \implies d = \{\max_i \{t_i\}\} \]

Examples of the $d_i$ multisets having many common elements:

\[ d_0 = \{5, 4, 3\} \quad d_1 = \{5, 4, 3\} \implies d = \{5, 4, 3\} \]
\[ d_0 = d_1 = \ldots d_{m-1} \implies d = d_0 \]
\[ d_0 = \{5, 4, 3\} \quad d_1 = \{5, 4, 2\} \quad d_2 = \{5, 4, 4\} \implies d = \{5, 4, 4\} \]
\[ d_0 = D \cup \{t_0\} \quad d_1 = D \cup \{t_1\} \quad \ldots \quad d_{m-1} = D \cup \{t_{m-1}\} \]
\[ \implies d = D \cup \{\max_i \{t_i\}\} \]

Consider a more general case in which $d$ is iteratively populated:

\[ d_0 = \{5, 4, 2\} \quad d_1 = \{5, 3, 3\} \quad d = \{\} \]

The intersection of the two multisets should certainly be in $d$, leaving:

\[ d_0 = \{4, 2\} \quad d_1 = \{3, 3\} \quad d = \{5\} \]

The maximum of these two multisets should certainly be in $d$ and can be used to implement $d_0$’s monomial of total degree four and $d_1$’s monomial of total degree three, leaving:

\[ d_0 = \{2\} \quad d_1 = \{3\} \quad d = \{5, 4\} \]

Again taking the maximum again results in $d = \{5, 4, 3\}$, conclude

\[ d_0 = \{5, 4, 2\} \quad d_1 = \{5, 3, 3\} \implies d = \{5, 4, 3\} \]
This process can be algorithmically captured as follows:

**Inputs** $d_0, d_1, \ldots, d_{m-1}$ — multisets of total degrees of $p_0, \ldots, p_{m-1}$

**Outputs** $d$ — multiset of total degrees of $p$

$t = d_0$

for $i = 1, \ldots, m - 1$ loop

$d = d_i \cap t$

$t = t \setminus d$

$d_i = d_i \setminus d$

while ($t \neq \{\}$) AND ($d_i \neq \{\}$) loop

$d = d \cup \{\max(\max(t), \max(d_i))\}$

$t = t \setminus \{\max(t)\}$

$d_i = d_i \setminus \{\max(d_i)\}$

endloop

$t = d$

endloop

return d

(4.4)

This algorithm uses multiset subtraction, intersection and sum. An example of multiset intersection is $\{1,1,1,3\} \cap \{1,1,2\} = \{1,1\}$ and multiset sum is $\{1,1\} \cup \{1,2\} = \{1,1,1,2\}$. Applying this algorithm to the case $d_0 = \{5,4,2\}, d_1 = \{5,3,3\}$ produces the following intermediate multisets:

\[
\begin{align*}
t &= \{5,4,2\} \\
d &= d_1 \cap t = \{5\} \quad t = t \setminus d = \{4,2\} \quad d_1 = d_1 \setminus d = \{3,3\} \\
d &= d \cup \{\max(\max(\{4,2\}), \max(\{3,3\}))\} = \{5,4\} \quad t = \{2\} \quad d_1 = \{3\} \\
d &= d \cup \{\max(\max(\{2\}), \max(\{3\}))\} = \{5,4,3\} \quad t = \{\} \quad d_1 = \{\} \\
d &= \{5,4,3\} \quad t = \{5,4,3\} \quad d_1 = \{\}
\end{align*}
\]

This result matches the expected result of $\{5,4,3\}$. This algorithm guarantees that the number of terms in $d$ is no greater than any of the number of terms in any of the $p_i$. Given this construction of the multiset $d$, the optimisation can be restated but now with a restriction on the number of
terms and total degrees of the monomials in $p$:

Given $p_i$, select $x_{j,i}$ and $p$ minimising

$$\sum_i \text{num of distinct elements in } \{x_{1,i}, x_{2,i}, \ldots, x_{m,i}\}$$

subject to $p(\tilde{x}(s, x)) = y(s, x)$ for all $s$ and $x$

and multiset of total degree of monomials in $p = d$

4.3.2. Solving the Optimisation Problem

Now that the number of terms and total degrees of the monomials in $p$ is fixed, the minimisation problem is now one of allocation. Consider the example:

$$y = (s == 0)?\ abcde + bcde + de :$$

$$(s == 1)?\ bcdea + bcd + cde :$$

$bcde + abc + bcd$$

Applying the Algorithm 4.4 which determines $d$, $p$ has total monomial degrees $d = \{5, 4, 3\}$, inserting ones such that the $p_i$ has the same monomial total degrees as in the multiset $d$:

$$y = (s == 0)?\ abcde + bcde + 1 \times de :$$

$$(s == 1)?\ bcdea + 1 \times bcd + cde :$$

$$1 \times bcde + 1 \times abc + bcd$$

A convenient representation is the following matrix form:

\[
\begin{pmatrix}
a & b & c & d & e \\
b & c & d & e & a \\
1 & b & c & d & e
\end{pmatrix}
\begin{pmatrix}
b & c & d & e \\
1 & b & c & d \\
1 & a & b & c
\end{pmatrix}
\begin{pmatrix}
1 & d & e \\
c & d & e \\
b & c & d
\end{pmatrix}
\]

These matrices have height $m$, their widths correspond to the elements in $d$ and there is a matrix for each monomial in $p$. Permutation of each row in each matrix leaves the polynomials the matrices represent unaltered due to the commutativity of multiplication. Further, the $i$th row of a matrix can be swapped with the $i$th row of another matrix due to the associativity of
addition. If the widths of the matrices differ a swap may be possible as long as the variables introduced or removed are ones. The number of distinct elements in each column corresponds to the implementation cost of selection required to form each $x_i$. These values for the current matrices are, along with their total which is the objective function of the optimisation problem:

\[
\begin{bmatrix}
  a & b & c & d & e \\
  b & c & d & e & a \\
  1 & b & c & d & e
\end{bmatrix}
\begin{bmatrix}
  b & c & d & e \\
  1 & b & c & d \\
  1 & a & b & c
\end{bmatrix}
\begin{bmatrix}
  1 & d & e \\
  c & d & e \\
  b & c & d
\end{bmatrix}
\]

Num. Distinct \(3 \ 2 \ 2 \ 2 \ 2\) \(2 \ 3 \ 3 \ 3\) \(3 \ 2 \ 2\) \(29\)

An example of an improved objective function value achieved by performing matrix row permutation and valid row swaps is:

\[
\begin{bmatrix}
  a & b & c & d & e \\
  a & b & c & d & e \\
  1 & b & c & d & e
\end{bmatrix}
\begin{bmatrix}
  e & b & c & d \\
  1 & b & c & d \\
  1 & b & c & d
\end{bmatrix}
\begin{bmatrix}
  1 & d & e \\
  c & d & e \\
  b & c & a
\end{bmatrix}
\]

Num. Distinct \(2 \ 1 \ 1 \ 1 \ 1\) \(2 \ 1 \ 1 \ 1\) \(3 \ 2 \ 2\) \(18\)

In order to produce a compact optimisation problem consider replacing the variables $a$, $b$, $c$, $d$ and $e$ with the first five prime numbers 2, 3, 5, 7 and 11 and augmenting each matrix with the product of each row:

\[
\begin{bmatrix}
  2 & 3 & 5 & 7 & 11 \\
  2 & 3 & 5 & 7 & 11 \\
  1 & 3 & 5 & 7 & 11
\end{bmatrix}
\begin{bmatrix}
  11 & 3 & 5 & 7 \\
  1 & 3 & 5 & 7 \\
  1 & 3 & 5 & 7
\end{bmatrix}
\begin{bmatrix}
  77 & 1 & 7 & 11 \\
  385 & 5 & 7 & 11 \\
  30 & 3 & 5 & 2
\end{bmatrix}
\]

\[
\begin{bmatrix}
  2 & 1 & 1 & 1 & 1 \\
  2 & 1 & 1 & 1 \\
  3 & 2 & 2
\end{bmatrix}
\]

It has been assumed that the monomials in a given $p_i$ are unique, hence, due to the uniqueness of prime factorisation, the products of primes per row will be unique. This allows for the recasting of the optimisation problem in terms of integers. Find integer matrices with entries from 1, 2, 3, 5, 7 and 11, such that row products are constrained to exactly match a set of integers. Of these matrices, minimise the sum of the number of distinct entries per column. Labeling these three matrices as $1A$, $2A$ and $3A$ and
their entries in the form $a_{j,k}$, the optimisation problem is now:

$$\min \sum_{i,k} \text{num of distinct elements over } j \text{ in } a_{j,k}$$

subject to

$$a_{j,k} \in \{1, 2, 3, 5, 7, 11\}$$

$$\left( \prod_k a_{1,k} \prod_k a_{2,k} \prod_k a_{3,k} \right) = (2310, 1155, 77)$$

$$\left( \prod_k a_{2,k} \prod_k a_{2,k} \prod_k a_{3,k} \right) = (2310, 105, 385)$$

$$\left( \prod_k a_{3,k} \prod_k a_{3,k} \prod_k a_{3,k} \right) = (1155, 105, 30)$$

The general integer program is then (using $q_1$, $q_2$, ..., $q_n$ to denote the first $n$ primes and $M(x_1, x_2, ..., x_n)$ to denote a particular monomial and $M(q_1, q_2, ..., q_n)$ to denote the value of that monomial when setting variable $x_j$ to be prime $q_j$):

$$\min \sum_{i,k} \text{num of distinct elements over } j \text{ in } a_{j,k}$$

subject to

$$a_{j,k} \text{ have height } m \text{ and have widths equal to the multiset } d$$

$$a_{j,k} \in \{1, q_1, q_2, ..., q_n\}$$

$$M(q_1, q_2, ..., q_n) \in \left\{ \prod_k a_{j,k} : i = 1, 2, ... \right\}$$

for all monomials $M(x_1, x_2, ..., x_n)$ in all $p_j$ (4.5)

This problem can be solved using off-the-shelf SMT solvers such as [165]. The solution to the example provided by the optimisation is:

$$\left( \begin{array}{cccc} b & d & c & e & a \\ b & d & c & e & a \\ b & 1 & c & 1 & a \end{array} \right) \left( \begin{array}{cccc} c & e & d & b \\ c & 1 & d & b \\ c & e & d & b \end{array} \right) \left( \begin{array}{cccc} 1 & e & d \\ c & e & d \\ c & b & d \end{array} \right)$$

Num. Distinct $\left( \begin{array}{cccc} 1 & 2 & 1 & 2 \\ 1 & 2 & 1 & 1 \\ 2 & 2 & 1 \end{array} \right)$ 17

94
Which corresponds to a rewriting of the form:

\[
\begin{align*}
c_1 &= (s == 0) ? 1 : c \\
d_1 &= (s \geq 2) ? 1 : d \\
e_1 &= (s \geq 2) ? 1 : e \\
e_2 &= (s == 1) ? 1 : e \\
e_3 &= (s \geq 2) ? b : e \\
y &= abcd_1e_1 + bcde_2 + c_1de_3
\end{align*}
\]

Summarising the complete procedure:

- The starting equation is (4.1) on page 79 which requires the mutually exclusive evaluation of \(p_0, p_1, \ldots, p_{m-1}\).
- Produce the multisets of total degrees of \(p_0, p_1, \ldots, p_{m-1}\), denoted \(d_0, d_1, \ldots, d_{m-1}\).
- Execute Algorithm 4.4 to produce the multiset of total degrees of polynomial \(p\).
- Solve the optimisation problem (4.5).
- Replace each prime entries, \(q_j\) of the matrices \(A\) with \(x_j\).
- The columns of the matrices are then the \(x_{1,i}, x_{2,i}, \ldots, x_{m,i}\) entries in equation (4.2).

4.3.3. Lifting the Restrictions

**Polynomials with differing numbers of terms** Algorithm 4.4 which produces the multiset of total degrees of \(p\) still functions if the polynomials \(p_i\) have differing numbers of terms. Consider the example:

\[
\begin{align*}
y &= (s == 0) ? \quad bcde + de : \\
     &\quad (s == 1) ? \quad bcdea + bcd + cde : \\
     &\quad \quad bcde + abc + bcd
\end{align*}
\]
Algorithm 4.4 now returns \( d = \{5, 3, 3\} \) and the resultant integer program is:

\[
\begin{align*}
\min & \quad \sum_{i,k} \text{num of distinct elements over } j \text{ in } iA_{j,k} \\
\text{subject to } & \quad iA_{j,k} \in \{0, 1, 2, 3, 5, 7, 11\} \\
& \quad \left( \prod_k 1_{A_{1,k}} \prod_k 2_{A_{1,k}} \prod_k 3_{A_{1,k}} \right) = (0, 1155, 77) \\
& \quad \left( \prod_k 1_{A_{2,k}} \prod_k 2_{A_{2,k}} \prod_k 3_{A_{2,k}} \right) = (2310, 105, 385) \\
& \quad \left( \prod_k 1_{A_{3,k}} \prod_k 2_{A_{3,k}} \prod_k 3_{A_{3,k}} \right) = (1155, 105, 30)
\end{align*}
\]

whose solution translates into the following polynomial rewrite:

\[
\begin{align*}
a_1 &= (s == 1)? a : 1 \\
c_1 &= (s == 0)? 1 : c \\
d_1 &= (s == 0)? 0 : (s == 1)? d : a \\
e_1 &= (s \geq 2)? b : e \\
y &= a_1bcde + c_1de_1 + bcd_1
\end{align*}
\]

In general, given that \( d_i \) is the multiset of total degrees of the polynomial \( p_i \), then \(|d| - |d_i|\) (note \(|.|\) denotes set size) is the difference between the number of monomials in \( p \) and \( p_i \). It is thus required to enhance the optimisation problem (4.5) by requiring that zero appears within the row products a
number of times equal to $|d| - |d_i|$: 

$$
\min \sum_{i,k} \text{num of distinct elements over } j \text{ in } iA_{j,k}
$$

subject to 

$iA$ have height $m$ and have widths equal to the multiset $d$

$iA_{j,k} \in \{0, 1, q_1, q_2, ..., q_n\}$

$M(q_1, q_2, ..., q_n) \in \left\{ \prod_k iA_{j,k} : i = 1, 2, ... \right\}$

for all monomials $M(x_1, x_2, ..., x_n)$ in all $p_j$

$$
\left| \left\{ \prod_k iA_{j,k} = 0 : i = 1, 2, ... \right\} \right| = |d| - |d_j| \text{ for all } j \quad (4.6)
$$

Note that this integer program has fewer or equal the number of constraints as optimisation (4.5).

**Integer Coefficients and Distinct Monomials** Turning to the case where the polynomials have positive integer coefficients such as:

$$
y = (s == 0)? 2abc + 3bcd + 3de : 
\quad (s == 1)? abc + 2bc + 5cd : 
\quad 3bcd + abc + 2bc
$$

the constants can be replaced by variables $2 \rightarrow f$, $3 \rightarrow g$ and $5 \rightarrow h$:

$$
y = (s == 0)? fabc + gabcd + gde : 
\quad (s == 1)? abc + fbc + hcd : 
\quad gabcd + abc + fbc
$$

This problem is now in a form which satisfies the initial restrictions. However, the objective function needs some modification. Selecting between constants is less expensive, from an implementation perspective, than selecting between variables. Thus the objective function should not distinguish between any of the constants (including zero and one). The modified optimisation problem where $p_j$ have been modified to replace constants with
primes $q_{n+1}, \ldots, q_{n+v}$ is then:

$$\min \sum_{i,k} \text{num of distinct elements } q_1 \ldots q_n \text{ over } j \text{ in } iA_{j,k}$$

subject to

$iA$ have height $m$ and have widths equal to the multiset $d$

$$iA_{j,k} \in \{0, 1, q_1, q_2, \ldots, q_n, q_{n+1}, \ldots, q_{n+v}\}$$

$$T(q_1, q_2, \ldots, q_n) \in \left\{ \prod_k iA_{j,k} : i = 1, 2, \ldots \right\}$$

for all terms $T(x_1, x_2, \ldots, x_n)$ in all $p_j$

$$\left| \left\{ \prod_k iA_{j,k} = 0 : i = 1, 2, \ldots \right\} \right| = |d| - |d_j| \text{ for all } j \quad (4.7)$$

Under the assumption that the polynomial with integer coefficients has been simplified then all the monomials will be distinct. This still holds once the constants have been replaced by variables.

### 4.3.4. Changing Variables

Having performed the selection logic minimisation via an integer program, subsequent manipulations are performed via algebraic geometry. Consider the design:

$$y = (s == 0)? d^2 + cd :$$

$$(s == 1)? b^2 + ba :$$

$$d^2 + ad$$

This can be written as a polynomial by splitting the select signal $s$ into the upper and lower bits $s_1$ and $s_0$:

$$y = (1 - s_1)(1 - s_0)(d^2 + cd) + (1 - s_1)s_0(b^2 + ba) + s_1(d^2 + ad)$$

The associated optimised $A$ matrices for this problem are:

$$\begin{pmatrix}
  d & c \\
  b & a \\
  d & a
\end{pmatrix} \quad \begin{pmatrix}
  d & d \\
  b & b \\
  d & d
\end{pmatrix}$$
The columns of which are used to create the $\tilde{x}$ variables.

\[ \tilde{x}_1 = (s == 0)?d : (s == 1)?b : d \]
\[ \tilde{x}_2 = (s == 0)?c : (s == 1)?a : a \]
\[ \tilde{x}_3 = (s == 0)?d : (s == 1)?b : d \]
\[ \tilde{x}_4 = (s == 0)?d : (s == 1)?b : d \] (4.8)

Writing these as polynomials:

\[ \tilde{x}_1 = (1 - s_1)(1 - s_0)d + (1 - s_1)s_0b + s_1d \]
\[ \tilde{x}_2 = (1 - s_1)(1 - s_0)c + (1 - s_1)s_0a + s_1a \]
\[ \tilde{x}_3 = (1 - s_1)(1 - s_0)d + (1 - s_1)s_0b + s_1d \]
\[ \tilde{x}_4 = (1 - s_1)(1 - s_0)d + (1 - s_1)s_0b + s_1d \] (4.9)

Using the elimination techniques described in the preliminaries chapter on page 70, elimination first requires creating the ideal:

\[ I = \langle \tilde{x}_1 - (1 - s_1)(1 - s_0)d - (1 - s_1)s_0b - s_1d, \]
\[ \tilde{x}_2 - (1 - s_1)(1 - s_0)c - (1 - s_1)s_0a - s_1a, \]
\[ \tilde{x}_3 - (1 - s_1)(1 - s_0)d - (1 - s_1)s_0b - s_1d, \]
\[ \tilde{x}_4 - (1 - s_1)(1 - s_0)d - (1 - s_1)s_0b - s_1d, \]
\[ s_1(1 - s_1), s_0(1 - s_0) \rangle \] (4.10)

All these polynomials equal zero, the first four are derived from equation (4.9) and the last two from $s_1$ and $s_0$ only taking the values zero and one. Using an elimination order which places $\tilde{x}_i$ smaller than $s_1$, $s_0$ and $a$, $b$, $c$ and $d$, the following reduction performs the elimination by first computing a Gröbner basis, $J$, for $I$ and then computing:

\[ p(\tilde{x}) = y \mod J = \tilde{x}_4^2 + \tilde{x}_2\tilde{x}_4 \]
Note that despite $\tilde{x}_1 = \tilde{x}_3 = \tilde{x}_4$, $p$ is only a function in one of these variables.

So the result of the change of variables is:

\[
\tilde{x}_2 = (s == 0) ? c : (s == 1) ? a : a
\]

\[
\tilde{x}_4 = (s == 0) ? d : (s == 1) ? b : d
\]

\[
y = \tilde{x}_2^2 + \tilde{x}_2 \tilde{x}_4
\]

It is expected that an industry standard logic synthesis tool will simplify the equations for $\tilde{x}_i$. In general the change of variable process is:

- Produce polynomial equations for the selection logic for each $\tilde{x}_i$ by using the bits of signal $s$, in the form of equation (4.9).

- Create an ideal from $\tilde{x}_i$ minus these polynomials and $s_i(1 - s_i)$ for each bit of $s$, in the form of equation (4.10).

- Compute a Gröbner basis, $J$, for $I$ with an elimination order placing $\tilde{x}_i$ variables smaller than all others.

- Compute $p = y \mod J$.

- If, for any variable $\tilde{x}_i$ found within $p$, the associated column of an $A$ matrix contains identical entries $x_j$, then replace $\tilde{x}_i$ in $p$ with $x_j$.

- Create selection logic for the $\tilde{x}_i$ signals found within $p$ from columns of the optimised $A$ matrices, in the form of equation (4.8).

- If any $x_i$ were originally constants, replace the $x_i$ with these constants.

### 4.4. Optimising the Optional Negations

As established in Section 4.1, optional negations within a polynomial incur increased implementation costs. This section explores how optional negation can be performed on the output or the inputs. Consider, for $s$ a one bit input, the example:

\[
y = s? - ab : ab
\]
Applying the procedure so far created results in:

\[ y = (1 - 2s)ab \]

Now the observation regarding optional negations in Section 4.2 which described the motivating example is now proven more rigorously. If the output bit width of \( y \) is \( w \) then:

\[
\begin{align*}
    y &= (1 - 2s)ab \mod 2^w \\
    &= (1 - 2s)ab - 2s(1 - s) + s2^w \mod 2^w \\
    &= (1 - 2s)ab - s(1 - 2s) - s + s2^w \mod 2^w \\
    &= (1 - 2s)(ab - s) + s(2^w - 1) \mod 2^w \\
    &= s(2^w - 1 - (ab - s)) + (1 - s)(ab - s) \mod 2^w \\
    &= s?\overline{ab-s} : ab - s \mod 2^w \quad // \text{bitwise inversion of } ab - s \\
    &= (ab - s) \oplus s \mod 2^w \quad // \text{bitwise XOR of } ab - s \text{ with } s
\end{align*}
\]

This optimisation over the finite ring results in computing a simple multiply-add. As noted in Section 4.1 the final XOR can be taken off the critical path. In conclusion, this optimisation has an implementation cost almost indistinguishable from a single multiplication. This optimisation holds for any polynomial:

\[
y = s? - p(x) : p(x) = (p(x) - s) \oplus s
\]

This optimisation allows for optional negation of \( p \) with negligible overhead. The most general polynomials which are unaltered by the process so far, are those where the monomials only differ in their sign, e.g.:

\[
\begin{align*}
    y &= (s == 0)? -abc + ab - cd : \\
        (s == 1)? abc + ab + cd : \\
        -abc - ab + cd \\
    y &= (-1 + 2s_0 - 2s_1s_0)abc + (1 - 2s_1)ab + (-1 + 2s_0 + 2s_1 - 2s_1s_0)cd \\
&\quad (4.11)
\end{align*}
\]
Note that the terms in brackets only ever take values ±1, hence:

\[
(-1 + 2s_0 - 2s_1s_0)^2 = 1 \\
(1 - 2s_1)^2 = 1 \\
(-1 + 2s_0 + 2s_1 - 2s_1s_0)^2 = 1
\]

Consider:

\[
(-1 + 2s_0 - 2s_1s_0)y = abc + (-1 + 2s_0 - 2s_1s_0)(1 - 2s_1)ab + \\
(-1 + 2s_0 - 2s_1s_0)(-1 + 2s_0 + 2s_1 - 2s_1s_0)cd
\]

= \[abc + (-1 + 2s_0 + 2s_1 - 2s_1s_0)ab + (1 - 2s_1)cd\]

This simplification can be achieved algorithmically by computing:

\[
(-1 + 2s_0 - 2s_1s_0) \mod <s_1(1 - s_1), s_0(1 - s_0)>
\]

Conclude that:

\[
y = (-1 + 2s_0 - 2s_1s_0)(abc + (-1 + 2s_0 + 2s_1 - 2s_1s_0)ab + (1 - 2s_1)cd)
\]

(4.12)

The optional negations can now be written as follows:

\[
S_1[0 : 0] = 1 - s_0 + s_1s_0 \\
S_2[0 : 0] = 1 - s_0 - s_1 + s_1s_0 \\
S_3[0 : 0] = s_1 \\
y = (1 - 2S_1)(abc + (1 - 2S_2)ab + (1 - 2S_3)cd) \\
y = (abc + (S_2? - a : a)b + (S_3? - c : c)d - S_1) \oplus S_1 \\
y = (abc + (a \oplus S_2)b + S_2b + (c \oplus S_3)d + S_3d - S_1) \oplus S_1
\]
Simplifying:

\[
\begin{align*}
S_1[0 : 0] &= 1 \oplus s_0 \oplus s_1 s_0 = \overline{s_0 s_1} \\
S_2[0 : 0] &= 1 \oplus s_0 \oplus s_1 \oplus s_1 s_0 = \overline{s_1 \vee s_0} \\
S_3[0 : 0] &= s_1 \\
y &= [abc + (a \oplus S_2)b + S_2b + (c \oplus S_3)d + S_3d - S_1] \oplus S_1
\end{align*}
\]

This formulation has now achieved a polynomial within the square brackets with all optional negations being handled by XORing on inputs.

In the general case the starting polynomial is of the type in equation (4.11) which is in the form

\[
y = \sum \alpha q_\alpha(s)x^\alpha
\]

for some polynomials \( q_\alpha \) which for all possible values of \( s \) evaluate to \( \pm 1 \). The implementation cost of the polynomial will be dominated by the monomials with the largest total degree. Optional negation of the entire polynomial has been shown to be relatively inexpensive to achieve. So factor out one of polynomials in \( s \), say \( q_\alpha^2 \neq 1 \), which is the coefficient of one of the monomials with largest total degree. It is always possible to perform this factorisation because the polynomials \( q \) evaluate to \( \pm 1 \) for all possible values of \( s \). This results in:

\[
y = q_\alpha(s)p'(x, s) \quad /\, / \text{ for some polynomial } p' \\
q_\alpha(s)y = p'(x, s) \quad /\, / \text{ as } q_\alpha^2 = 1
\]

To calculate \( p' \), first create the Gröbner basis, \( J \) of the ideal, \( I \):

\[
I = \langle s_0(1 - s_0), s_1(1 - s_1), ... \rangle
\]

where \( s_i \) are each of the bits of \( s \). Then, for some \( q' \):

\[
p' = q_\alpha(s)y \mod J = \sum \alpha q'_\alpha(s)x^\alpha
\]

Each \( q \) polynomial takes the value \( \pm 1 \), so are of the form \( 1 - 2S_\alpha \) for some bits \( S_\alpha \). Compute these \( S_\alpha \):

\[
S_\alpha[0 : 0] = \frac{1}{2} (1 - q_\alpha(s))
\]

Also the factored polynomial \( q_\alpha(s) \) also takes the value \( \pm 1 \) and can be
written as $1 - 2S$, the value of $S$ is thus:

$$S[0 : 0] = \frac{1}{2} (1 - q_{\overline{a}}(s))$$

Note that these polynomials will always be of the following form:

$$\sum_{\beta} (\pm 1) \left( s_{0}^{\beta_{0}} s_{1}^{\beta_{1}} \ldots \right)$$

$$= \sum_{\beta} (\pm 1) \left( s_{0}^{\beta_{0}} s_{1}^{\beta_{1}} \ldots \right) \mod 2$$

$$= \bigoplus_{\beta} \left( s_{0}^{\beta_{0}} \land s_{1}^{\beta_{1}} \ldots \right)$$

**e.g.** $S = s_{0} + s_{1} - s_{0}s_{1} = s_{0} \oplus s_{1} \oplus (s_{0} \land s_{1}) = s_{0} \lor s_{1}$

It is expected that an industry leading synthesis tool can efficiently simplify these expressions. Rewriting $y$ in terms of these variables:

$$y = (1 - 2S) \sum_{\alpha} \left( 1 - 2S_{\alpha} \right) x_{\alpha}$$

$$= \left( \left( \sum_{\alpha} \left( 1 - 2S_{\alpha} \right) x_{\alpha} \right) - S \right) \oplus S$$

Attention is now turned to the optional negations required by the signals $S_{\alpha}$. As discussed in Section 4.1, performing optional negation as part of polynomial evaluation increases implementation cost. These optional negations can be performed on the inputs to the polynomial. So an arbitrary variable from each monomial $x_{\alpha}$ can be chosen to be negated, denote this by $opneg_{\alpha}$:

$$y = \left( \left( \sum_{\alpha} \left( 1 - 2S_{\alpha} \right) x_{\alpha} \right) - S \right) \oplus S$$

$$= \left( \left( \sum_{\alpha} \left( (-1)^{S_{\alpha} \oplus opneg_{\alpha}} \frac{x_{\alpha}}{opneg_{\alpha}} \right) - S \right) \oplus S \right)$$

$$= \left( \left( \sum_{\alpha} \left( opneg_{\alpha} \oplus S_{\alpha} + S_{\alpha} \right) \frac{x_{\alpha}}{opneg_{\alpha}} \right) - S \right) \oplus S$$

This is the desired final form. To summarise the process:
• The starting polynomial is of the form $y(x, s) = \sum_{\alpha} q_{\alpha}(s)x^{\alpha}$.

• Compute $J$, the Gröbner basis of the ideal formed from each bit of $s$:
  $I = \langle s_0(1 - s_0), s_1(1 - s_1), \ldots \rangle$.

• Choose $q_{\alpha}(s)$ then compute
  
  \[
  p' = q_{\alpha}(s)y \mod J = \sum_{\alpha} q_{\alpha}'(s)x^{\alpha}
  \]
  
  \[
  S = \frac{1}{2} \left( 1 - q_{\alpha}(s) \right)
  \]
  
  \[
  S_{\alpha} = \frac{1}{2} \left( 1 - q_{\alpha}'(s) \right) \quad \text{for all } \alpha
  \]

• For each $S_{\alpha}$ the associated monomial must be optionally negated depending on this signal. This can be achieved by optionally negating any of the inputs to the monomial. So an arbitrary input variable to the monomial $x^{\alpha}$ is chosen to optionally negate, call this input $opnegx_{\alpha}$. Given the inputs were assumed to be of equal bitwidth, these can be arbitrarily chosen to be the leading variable within the monomial. The final desired form is then:

  \[
  opnegx_{\alpha}' = opnegx_{\alpha} \oplus S_{\alpha}
  \]
  
  \[
  y = \left( \left( \sum_{\alpha} (opnegx_{\alpha}' + S_{\alpha}) \frac{x^{\alpha}}{opnegx_{\alpha}} \right) - S \right) \oplus S
  \]

4.5. Overall Flow

The starting point of the overall flow is a set of $m$ mutually exclusive polynomials with integer coefficients:

\[
y = \begin{cases} 
  (s == 0)? \quad p_0(x_1, x_2, \ldots, x_n) & : \\
  (s == 1)? \quad p_1(x_1, x_2, \ldots, x_n) & : \\
  \vdots & \\
  (s == m - 2)? \quad p_{m-2}(x_1, x_2, \ldots, x_n) & : \quad p_{m-1}(x_1, x_2, \ldots, x_n)
\end{cases}
\]
The result of the flow is to transform this equation into the following form, where \( x_{i,j} \) are constant integers or one of \( x_1, x_2, ..., x_n \):

\[
\tilde{x}_1 = (s == 0)? x_{1,1} : (s == 1)? x_{2,1} : \cdots : x_{m,1} \\
\tilde{x}_2 = (s == 0)? x_{1,2} : (s == 1)? x_{2,2} : \cdots : x_{m,2} \\
\vdots \\
\tilde{x}_r = (s == 0)? x_{1,r} : (s == 1)? x_{2,r} : \cdots : x_{m,r} \\
y = p(\tilde{x}_1, \tilde{x}_2, ..., \tilde{x}_r)
\]

- **Precondition** \( p_i \) Expand and simplify \( p_i \) into sums of monomials (this is a well defined operation). Transform the \( p_i \) such that their coefficients are only \( \pm 1 \) by introducing new variables, \( x_{n+1}, x_{n+2}, ..., x_{n+v} \), for each distinct integer with non unit magnitude.

- **Calculate the total degrees of polynomial** \( p \) Create the multisets, \( d_i \), of total degrees of \( p_i \). Use Algorithm 4.4 to establish the total degrees of \( p \).

- **Solve the integer program for control minimisation** Solve the optimisation problem (4.7) using the first \( n + v \) primes \( q_1, q_2, ..., q_{n+v} \). In the resultant \( A \) matrices, replace \( q_i \) by variables \( x_i \). Each column of the \( A \) matrices form the entries \( x_{1,i}, x_{2,i}, ..., x_{m,i} \) which then create the variable \( \tilde{x}_i \).

- **Create the polynomial** \( p \) Create a Gröbner basis, \( J \), for ideal \( I \) using a term order which places \( \tilde{x} \) variables smaller than \( x \) variables. Ideal \( I \) is defined as follows using the bits of \( s \):

\[
I = \langle \tilde{x}_1 - ((s == 0)? x_{1,1} : (s == 1)? x_{2,1} : \cdots : x_{m,1}), \\
\vdots \\
\tilde{x}_r - ((s == 0)? x_{1,r} : (s == 1)? x_{2,r} : \cdots : x_{m,r}), \\
s_0(1 - s_0), s_1(1 - s_1), ... \rangle
\]

Then \( p = y \mod J \). If \( p \) involves any \( \tilde{x}_i \) which equal \( x_j \) for some \( j \) (which would arise if a column of an \( A \) matrix had identical entries) then replace \( \tilde{x}_i \) by \( x_j \). Replace \( x_{n+1}, x_{n+2}, ..., x_{n+v} \) by their original constants.
• **Factorise** \( p \)  
\( p \) is now treated as a polynomial with monomials in \( x_i \) and \( \tilde{x}_j \) with coefficients which are polynomials in \( s_k \). Choose a monomial of largest total degree whose coefficient is dependent on \( s \), call this coefficient \( q_{\alpha}(s) \). Calculate, where \( J' \) is the Gröbner basis of the ideal \( I' =< s_0(1 - s_0), s_1(1 - s_1), ... > \):

\[
p' = q_{\alpha}(s)p \mod J'
\]

Then \( p = q_{\alpha}(s)p' \).

• **Calculate the optional negation signals**  
If \( q'_{\alpha}(s) \) are the coefficients of \( p' \) then calculate:

\[
S = \frac{1}{2} (1 - q_{\alpha}(s)) \\
S_{\alpha} = \frac{1}{2} (1 - q'_{\alpha}(s))
\]

Then:

\[
p = (1 - 2S) \sum_{\alpha} (1 - 2S_{\alpha}) x_{\alpha}
\]

• **Produce the final form of** \( p \)  
Defining \( opnegx_{\alpha} \) as the largest variable in the monomial \( x_{\alpha} \) then the final form of \( p \) is:

\[
opnegx'_{\alpha} = opnegx_{\alpha} \oplus S_{\alpha} \quad \text{for all } \alpha
\]

\[
p = \left( \left( \sum_{\alpha} (opnegx'_{\alpha} + S_{\alpha}) \frac{x_{\alpha}}{opnegx_{\alpha}} \right) - S \right) \oplus S
\]

• **Produce optimised design**  
The equations defining \( \tilde{x}_i \), \( S \), \( S_{\alpha} \), \( opnegx'_{\alpha} \) and \( p \) define the optimised design.

### 4.6. Formal Verification

The overall flow described in the previous section, if placed in a production level environment, would use a variety of software packages and scripts. It is thus crucial that verification techniques exist that can prove formal equivalence between the pre and post optimised designs. However, naïve
formal verification of these optimisations is well beyond the capacity of current existing tools. As can be seen in the experiments section, an application of existing industry tools to these problems returns inconclusive results. Equivalence of polynomials over integer rings has been successfully explored in [150], [151] and [152] and even expanded to non polynomial operators in [167]. This section builds on this work by introducing an augmented use, or super usage, of existing industry verification tools to achieve formal verification of the optimisation presented in this chapter.

If the original problem statement is embodied in \( y_1(s, x) \) and the optimised form in \( y_2(s, x) \), the output bit width is \( n \), \( x_i \) have bitwidths \( n_i \) and \( s \) has bit width \( v \), then the formal equivalence problem is:

\[
y_1(s, x_1, x_2, ...) - y_2(s, x_1, x_2, ...) = 0 \mod 2^n
\]

for all \( s \in [0, 1, ..., 2^v - 1] \) and \( x_i \in [0, 1, ..., 2^{n_i} - 1] \) for all \( i \)

Given the problem formulation, it is known that \( y_i \) are polynomials in unsigned integer inputs \( x_i \) and the bits of \( s, s_0, s_1, ..., s_{v-1} \) with integer coefficients. In general, a datapath function is said to be polynomial if its inputs are unsigned integers and the function can be written as a polynomial with integer coefficients reduced modulo \( 2^n \), where \( n \) is the output bit-width. A variety of common operators and number formats are polynomial in nature. For example, for \( n \) bit integer inputs \( a \) and \( b \):

- **Signed Number**
  \[a\] = \(-2^{n-1} a[n-1] + a[n-2 : 0]\)

- **Signed Magnitude**
  \[a\] = \((-1)^{a[n-1]} a[n-2 : 0] = (1 - 2a[n-1])a[n-2 : 0]\)

- **Muxing**
  \[s?a : b = sa + (1-s)b\]

- **Bitwise Inversion**
  \[\overline{a} = 2^n - 1 - a\]

- **Left Shift**
  \[a << b = a \left( 2^{2^{n-1} - 1} b[n-1] + 1 \right) ... (3b[1] + 1)(b[0] + 1)\]

Every datapath function is polynomial in its input bits, due to the universality of NANDs and:

\[NAND(a, b) = 1 - ab\]

Recalling, from the preliminaries, the result on multivariate polynomial
equivalence over rings, equation (3.3) on page 78:

\[ p(x) = 0 \mod 2^n \quad \text{for all } x \in \mathbb{Z}^k \iff p(x) = 0 \mod 2^n \quad \text{for } x \in [0, 1, 2, \ldots, SF(2^n) - 1]^k \]  

(4.13)

This shows that it is not necessary to check all possible input values for \( x_i \) but an exponentially smaller number; just the first \( SF(2^n) \). In the worst case, a function may be polynomial in only its input bits, in which case the left and right hand side of equation (4.13) are identical and no reduction has been achieved. In order to apply equation (4.13) the polynomial nature of function must be determined, namely, in which inputs, parts of inputs or bits of inputs is the function polynomial. This can be done without having to form the polynomial, but by inspecting the operators. Consider the following datapath function with unsigned 16 bit inputs \( a, b, c \) and \( d \):

\[
\begin{align*}
t_0[16 : 0] &= c[15 : 0] + 1 \\
t_1[31 : 0] &= b[15 : 0] - t_0 \\
t_2[31 : 0] &= d[15 : 0] \ll t_0 \\
t_3[31 : 0] &= a[15 : 0]t_1 \\
y[31 : 0] &= t_3 + t_2
\end{align*}
\]  

(4.14)

A directed graph whose vertices are the inputs, outputs, and operators of this design can be created, Figure 4.3 shows the resultant data flow graph (DFG). The edges of the directed graph correspond to the interconnecting signals of the design, connecting inputs, outputs, and operators as necessary, with the appropriate direction. To establish the polynomial nature of this design, first note the polynomial behaviour of each operator node, demonstrated in Figure 4.4, where the \( P \) and \( N \) labels stand for polynomial and non polynomial respectively. Multiplication and addition are polynomial in both inputs. Left shift is polynomial in the shifted value but non polynomial in the shift value. Using the polynomial nature of each operator we can build up the polynomial nature of the entire design. The algorithm that establishes the polynomial nature of the inputs is in Figure 4.5.
Figure 4.3.: Example Data-Flow Graph.

Figure 4.4.: Polynomial Behaviour of Operators (P=Polynomial and N=Non-Polynomial).
**Inputs** Data Flow Graph and pre calculated operator polynomial behaviour

**Output** Labeled Data Flow Graph and Inputs

**Step 1:** Label the outputs of the DFG as $P$

**Step 2:** For every node $v$ for which all outputs are labeled

- If all outputs of $v$ are labeled $P$
- Then label the inputs as per the known operator’s polynomial behaviour
- Else label all the inputs as $N$

**Step 3:** Repeat Step 2 until all edges are labeled

**Step 4:** If all edges from an input are labeled $P$

- Then label the input $P$
- Else label the input $N$

Figure 4.5.: Algorithm for Establishing Polynomial Nature of Inputs.

Applying Algorithm 4.5 to the design in Figure 4.3 results in Figure 4.6:

Figure 4.6.: Augmented Data-Flow Graph.
In this case, the design is polynomial with respect to inputs $a$, $b$, and $d$.

### 4.6.1. The Waterfall Verification

The verification method requires the datapath to be extractable into a DFG with unsigned inputs and there must be no bit slicing of the inputs, *i.e.* *unbroken* inputs. Moreover the DFG does not contain internal bit widths, hence these must not affect functionality. So the first step in a stepped, or *waterfall*, verification would be between an original design $A$, and a modified design $A'$ which has unsigned and unbroken inputs and internal bit widths which are the same as the output width. So, for example in the case of the design in Figure 4.3 the following design would be derived:

\[
\begin{align*}
t_0[31 : 0] &= c[15 : 0] + 1 \\
t_1[31 : 0] &= b[15 : 0] - t_0 \\
t_2[31 : 0] &= d[15 : 0] << t_0 \\
t_3[31 : 0] &= a[15 : 0]t_1 \\
y[31 : 0] &= t_3 + t_2
\end{align*}
\] (4.15)

This design has a 32 bit output and is polynomial in the 16 bit inputs $a$, $b$, and $d$ and in all the inputs bits of input $c$. Using the result in equation (4.13) it is only required to check the first $SF(2^{32}) = 34$ values on each of the inputs in which the design is polynomial. This can be achieved by restricting the sizes of $a$, $b$ and $d$ to 6 bits in length. Having restricted the bit widths, the design is now:

\[
\begin{align*}
t_0[31 : 0] &= c[15 : 0] + 1 \\
t_1[31 : 0] &= b[5 : 0] - t_0 \\
t_2[31 : 0] &= d[5 : 0] << t_0 \\
t_3[31 : 0] &= a[5 : 0]t_1 \\
y[31 : 0] &= t_3 + t_2
\end{align*}
\] (4.16)

Figure 4.7 illustrates the complete verification process when formally verifying design $A$ against $B$. 

112
4.6.2. Overall Verification Flow

The precise steps of the verification methodology verifying designs $A$ and $B$ are:

- Produce designs $A'$ and $B'$. These designs have unsigned inputs, all internal signals are of bit width $n$ and no inputs are bit sliced throughout the design. If a signal is sliced during the design, then two inputs should be created. If an input is in floating point format, then it will need to be split into sign, exponent and mantissa. Sign magnitude inputs will need their most significant bit separated into a new input. Two's complement signed numbers will similarly need their most significant bit separated into a new input. If the inputs to $A'$ and $B'$ now differ, the fewest number of new inputs are created such that no bit splicing occurs in the two designs. A formal verification using standard tools will then be performed between $A$ and $A'$ as well as between $B$ and $B'$. If either of these fail then the method is not applicable to the given verification.

- Create a DFG from $A'$ and $B'$, apply Algorithm 4.5 to each DFG. The inputs which have been labeled as $P$ on both DFGs are then defined as polynomial.

- For each polynomial input with width $w_j$ compute, where $n$ is the
maximum output width:

\[ \lambda_j = \min \left( w_j, \lceil \log_2 (SF(2^n)) \rceil \right) \]

Note that \( SF(2^n) \) can be efficiently calculated via equation (3.1) on page 75 from the preliminaries chapter:

\[ SF(2^k) = \min \{ n : n - \text{Hamm}(n) \geq k \} \]

- Create designs \( A'' \) and \( B'' \) which are identical to \( A' \) and \( B' \) except for the fact that the polynomial inputs are reduced in size to their corresponding width \( \lambda_j \). If the verification between \( A'' \) and \( B'' \) succeeds then the designs \( A \) and \( B \) are formally equivalent, otherwise they are not.

Completing the example introduced in equation (4.14), consider designs \( A \) and \( B \):

<table>
<thead>
<tr>
<th>Design A</th>
<th>Design B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t0[16 : 0] = c[15 : 0] + 1 )</td>
<td>( t0[16 : 0] = c[15 : 0] + 1 )</td>
</tr>
<tr>
<td>( t1[31 : 0] = b[15 : 0] - t0 )</td>
<td>( t1[31 : 0] = d[15 : 0] &lt;&lt; c[15 : 0] )</td>
</tr>
<tr>
<td>( t2[31 : 0] = d[15 : 0] &lt;&lt; t0 )</td>
<td>( t2[31 : 0] = a[15 : 0]b[15 : 0] )</td>
</tr>
<tr>
<td>( t3[31 : 0] = a[15 : 0]t1 )</td>
<td>( t3[31 : 0] = a[15 : 0]t0 )</td>
</tr>
<tr>
<td>( y[31 : 0] = t3 + t2 )</td>
<td>( y[31 : 0] = t2 - t3 + 2t1 )</td>
</tr>
</tbody>
</table>

The derived designs \( A' \) and \( B' \) are:

<table>
<thead>
<tr>
<th>Design ( A' )</th>
<th>Design ( B' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t0[31 : 0] = c + 1 )</td>
<td>( t0[31 : 0] = c + 1 )</td>
</tr>
<tr>
<td>( t1[31 : 0] = b - t0 )</td>
<td>( t1[31 : 0] = d &lt;&lt; c )</td>
</tr>
<tr>
<td>( t2[31 : 0] = d &lt;&lt; t0 )</td>
<td>( t2[31 : 0] = ab )</td>
</tr>
<tr>
<td>( t3[31 : 0] = at1 )</td>
<td>( t3[31 : 0] = at0 )</td>
</tr>
<tr>
<td>( y[31 : 0] = t3 + t2 )</td>
<td>( y[31 : 0] = t2 - t3 + 2t1 )</td>
</tr>
</tbody>
</table>

The DFG of \( A' \) can be found in Figure 4.3. By applying Algorithm 4.5 to the DFG the polynomial inputs were found to be \( a, b \) and \( d \). The output bit width in this case is 32 so it is required to compute \( SF(2^{32}) \). Recall
equation (3.2) from the preliminaries chapter:

\[ SF\left(2^{2^k}\right) = 2^k + 2 \]

Hence \( SF(2^{32}) = 34 \), calculating \( \lambda \):

\[ \lambda_j = \min\left(16, \lceil \log_2\left(SF(2^{32})\right) \rceil\right) = 6 \]

Now deriving the designs with reduced bit width, \( A'' \) and \( B'' \):

<table>
<thead>
<tr>
<th>Design ( A'' )</th>
<th>Design ( B'' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_0[31 : 0] = c[15 : 0] + 1 )</td>
<td>( t_0[31 : 0] = c[15 : 0] + 1 )</td>
</tr>
<tr>
<td>( t_1[31 : 0] = b[5 : 0] - t_0 )</td>
<td>( t_1[31 : 0] = d[5 : 0] \ll c[15 : 0] )</td>
</tr>
<tr>
<td>( t_2[31 : 0] = d[5 : 0] \ll t_0 )</td>
<td>( t_2[31 : 0] = a[5 : 0] b[5 : 0] )</td>
</tr>
<tr>
<td>( t_3[31 : 0] = a[5 : 0] t_1 )</td>
<td>( t_3[31 : 0] = a[5 : 0] t_0 )</td>
</tr>
<tr>
<td>( y[31 : 0] = t_3 + t_2 )</td>
<td>( y[31 : 0] = t_2 - t_3 + 2 t_1 )</td>
</tr>
</tbody>
</table>

Formally verifying \( A'' \) against \( B'' \) is a much simpler verification than \( A \) against \( B \). \( SF(2^n) \) is of order \( n \), thus replacing exponential complexity with linear.
4.7. Experiments

The overall flow presented in Section 4.5 provides an algorithm for optimising the implementation of a mutually exclusive set of polynomials. The process requires standard operations in computer algebra such as expansion of polynomials with respect to a particular monomial ordering, Gröbner basis calculation, factorisation and reduction modulo an ideal. These operations have been discussed in Chapter 3 and can be phrased in a computer algebra system such as [43]. In addition to these operations the solution of an integer program is required, this can be phrased and solved using a tool such as [165]. The flow can thus be made fully automatic, however the engineering effort required is beyond the scope of this thesis. In light of this, the experiments performed necessitated the selection of only a small number of examples.

This section demonstrates that the benefits seen in the motivating example in Section 4.2 are not restricted to one example and that the proposed flow offers considerable area reduction benefits in a variety of other cases. To demonstrate the efficacy of the flow, examples that target the individual techniques that contribute to the flow were created as well as general examples which illustrate the impact of the entire flow.

The technique used to compare the effects of the transformations was to take the pre and post optimised designs and perform logic synthesis using Synopsys Design Compiler 2012.06-SP2 [162] in ultra mode using the TSMC 40nm library Tcnn40lpbwp. The synthesis tool was requested to synthesise the designs to achieve different delays. By applying Boolean optimization techniques and utilizing different standard cells, Design Compiler seeks the design with smallest area that meets the required delay. Thus the full delay and area trade off of the various designs can be seen. These curves are generally monotonically decreasing in nature, as for a larger delay the tool has more freedom to minimise the area cost, the points with smallest delay will have the largest area. For each experiment, the formal verification of the transformation was performed with leading industry tools as well as by using the technique put forward in Section 4.6 and runtimes are reported. The integer program was solved using Synopsys Hector [165] and where appropriate the runtimes of the integer program are also presented. The data inputs for all these benchmarks are unsigned 16 bit variables.
First consider the motivating example of Section 4.2, the optimisation presented there attempted to make the cost of implementing a set of mutually exclusive SOPs close to that of $AB + C$. The area-delay curves for the pre and post optimised design as well as $AB + C$ can be found in Figure 4.8. The optimised design $m_5$ exhibits a maximum area reduction of 68% over the original and approaches the implementation cost of $AB + C$ which is up to 75% smaller. Thus the optimisation approaches its goal of mitigating the costs incurred by the introduction of mutually exclusive polynomials and optional negations.

![Area-Delay curves for $m_1$, $m_5$ and $AB + C$ which forms a lower bound on achievable area.](image)

**Figure 4.8.:** Area-Delay curves for $m_1$, $m_5$ and $AB + C$ which forms a lower bound on achievable area.

### 4.7.1. Control Logic Minimisation Experiments

In this section, a selection of designs is considered which are designed to stress and illustrate the impact of the control logic minimisation technique presented in this chapter. Designs which highlight the effect of control logic minimisation are designs where the $p_i$ are identical up to reordering in their
terms and products. An example of such a design as is follows:

\[
y_{1,0} = (s == 0)? \ ab + cd + ef + gh : \\
(s == 1)? \ bc + de + fg + ha : \\
(s == 2)? \ cd + ef + gh + ab : \\
(s == 3)? \ de + fg + ha + bc : \\
(s == 4)? \ ef + gh + ab + cd : \\
(s == 5)? \ fg + ha + bc + de : \\
(s == 6)? \ gh + ab + cd + ef : \\
\]

\[
ha + bc + de + fg \\
\]

(4.17)

Merging all these polynomials together without performing the control logic minimisation results in the following design:

\[
a' = (s == 0)?a : (s == 1)?b : (s == 2)?c : \ldots (s == 6)?g : h \\
b' = (s == 0)?b : (s == 1)?c : (s == 2)?d : \ldots (s == 6)?h : a \\
\ldots \\
h' = (s == 0)?h : (s == 1)?a : (s == 2)?b : \ldots (s == 6)?f : g \\
y_{1,1} = a'b' + c'd' + e'f' + g'h' \\
\]

However each of the mutually exclusive polynomials contains a permutation of the input variables, which has the implication that there are really only two mutually exclusive polynomials present and thus the design is essentially only of the following form:

\[
s_0? \ cb + ed + gf + ah : \ ab + cd + ef + gh \\
\]

118
As such, after applying the control logic minimisation the flow presented in Section 4.5 the result is as follows:

\[
\begin{align*}
\tilde{x}_1 &= s_0?c : a \\
\tilde{x}_2 &= s_0?e : c \\
\tilde{x}_3 &= s_0?g : e \\
\tilde{x}_4 &= s_0?a : g \\
y_{1,2} &= \tilde{x}_1b + \tilde{x}_2d + \tilde{x}_3f + \tilde{x}_4h
\end{align*}
\]

This provides the most dramatic illustration of the reduction in muxing provided by the control logic minimisation technique. Using a random assignment of variables \(a\) to \(h\) in the example found in equation (4.17) can provide further examples of the potential benefit of the control logic minimisation. The following two designs have random variable assignments:

\[
\begin{align*}
y_{2,0} = (s == 0)? & \text{ec + hh + dg + ac :} \\
&s == 1)? & \text{ef + dg + he + af :} \\
&s == 2)? & \text{ba + af + ce + bg :} \\
&s == 3)? & \text{df + ce + ad + ea :} \\
&s == 4)? & \text{cb + hh + fe + fb :} \\
&s == 5)? & \text{he + cg + ag + ad :} \\
&s == 6)? & \text{bg + bb + af + gf :} \\
& & \text{fh + bg + ag + da}
\end{align*}
\]

\[
\begin{align*}
y_{3,0} = (s == 0)? & \text{fe + ca + ah + gb :} \\
&s == 1)? & \text{cg + de + gc + hg :} \\
&s == 2)? & \text{gd + fa + ga + ef :} \\
&s == 3)? & \text{hd + ha + db + gh :} \\
&s == 4)? & \text{bf + cb + fg + db :} \\
&s == 5)? & \text{af + ga + eb + af :} \\
&s == 6)? & \text{hd + ba + fd + bg :} \\
& & \text{gh + hg + eb + ce}
\end{align*}
\]
Merging these polynomials together without and with applying the control logic minimisation technique results in designs $y_{2,1}$ and $y_{2,2}$ respectively from the first random assignment and $y_{3,1}$, $y_{3,2}$ for the second.

$$a' = (s == 0)?e : (s == 1)?e : (s == 2)?b : \ldots (s == 6)?b : f$$
$$b' = (s == 0)?c : (s == 1)?f : (s == 2)?a : \ldots (s == 6)?g : h$$
$$\ldots$$
$$h' = (s == 0)?e : (s == 1)?f : (s == 2)?g : \ldots (s == 6)?f : a$$
$$y_{2,1} = a'b' + c'd' + e'f' + gh'$$

$$a' = (s == 0)?h : (s == 1)?f : (s == 2)?f : (s == 3)?d :$$
$$\quad (s == 4)?h : (s == 5)?d : (s == 6)?f : d$$
$$b' = (s == 0)?h : (s == 1)?a : (s == 2)?a : (s == 3)?a :$$
$$\quad (s == 4)?h : (s == 5)?a : (s == 6)?a : a$$
$$c' = (s == 0)?d : (s == 1)?d : (s == 2)?b : (s == 3)?d :$$
$$\quad (s == 4)?b : (s == 5)?c : (s == 6)?b : b$$
$$d' = (s == 0)?g : (s == 1)?g : (s == 2)?g : (s == 3)?f :$$
$$\quad (s == 4)?f : (s == 5)?g : (s == 6)?g : g$$
$$e' = (s == 0)?e : (s == 1)?e : (s == 2)?e : (s == 3)?e :$$
$$\quad (s == 4)?b : (s == 5)?c : (s == 6)?b : f$$
$$f' = (s == 0)?c : (s == 1)?h : (s == 2)?c : (s == 3)?c :$$
$$\quad (s == 4)?c : (s == 5)?h : (s == 6)?b : h$$
$$g' = (s == 0)?e : (s == 1)?e : (s == 2)?b : (s == 3)?e :$$
$$\quad (s == 4)?e : (s == 5)?g : (s == 6)?g : g$$
$$h' = (s == 0)?a : (s == 1)?f : (s == 2)?a : (s == 3)?a :$$
$$\quad (s == 4)?f : (s == 5)?a : (s == 6)?f : a$$
$$y_{2,2} = a'b' + c'd' + e'f' + gh'$$
\[ a' = (s = 0)? f : (s = 1)? c : (s = 2)? g : \ldots (s = 6)? h : g \]
\[ b' = (s = 0)? e : (s = 1)? g : (s = 2)? d : \ldots (s = 6)? d : h \]
\[ h' = (s = 0)? b : (s = 1)? g : (s = 2)? f : \ldots (s = 6)? g : e \]
\[ y_{3,1} = a'b' + c'd' + e'f' + g'h' \]

The area-delay curves for each of these three designs, with and without the application of the control logic minimisation can be found in Figures 4.9, 4.10 and 4.11. Where Figure 4.9 compares designs \( y_{1,1} \) and \( y_{1,2} \), Figure 4.10 compares designs \( y_{2,1} \) and \( y_{2,2} \) and Figure 4.11 compares designs \( y_{3,1} \) and \( y_{3,2} \).
Figure 4.9.: Area-Delay curves for $y_{1,1}$ and $y_{1,2}$.

Figure 4.10.: Area-Delay curves for $y_{2,1}$ and $y_{2,2}$. 
Table 4.2.: Integer Program Runtimes.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_{1.2}$</td>
<td>60</td>
</tr>
<tr>
<td>$y_{2.2}$</td>
<td>99</td>
</tr>
<tr>
<td>$y_{3.2}$</td>
<td>151</td>
</tr>
</tbody>
</table>

Figure 4.11.: Area-Delay curves for $y_{3.1}$ and $y_{3.2}$.

These figures demonstrate that the permutation example first introduced with area-delay curves as found in Figure 4.9 demonstrate a 13% speed improvement as well as an area reduction of between 18 and 32%. Figures 4.10 and 4.11 derive from random variable allocation and, as a result, the area benefit is not as pronounced with a percentage area decrease of between 6 and 16% in both cases. The integer program takes order of minutes to complete these optimisations as seen from Table 4.2.

The optimality of the integer program and the strict minimisation of the associated muxing logic that results means that the control logic minimisation technique presented will always contribute to a reduction in hardware implementation costs. These experiments have shown that in isolation these benefits alone can be up to 32% in magnitude.
4.7.2. Optional Negation Experiments

One of the crucial optimisations presented in Section 4.4 was how to perform the optional negation of a polynomial:

\[ s? \cdot p : p = (p - s) \oplus s \]

To demonstrate the efficacy of this optimisation consider the following designs where it is required to implement either a polynomial or its negation:

\[
\begin{align*}
y_1 &= s? - ab : ab = (ab - s) \oplus s \\
y_2 &= s? - ab - cd : ab + cd = (ab + cd - s) \oplus s \\
y_3 &= s? - ab - cd - ef : ab + cd + ef = (ab + cd + ef - s) \oplus s \\
y_4 &= s? - ab - cd - ef - gh : ab + cd + ef + gh \\
&= (ab + cd + ef + gh - s) \oplus s
\end{align*}
\]

Synthesis of these designs establishes the benefits of the optimisation with respect to the size of the polynomial in question. The area-delay curves for each of the designs with and without the optimisation can be found in Figures 4.12, 4.13, 4.14 and 4.15.
Figure 4.12.: Area-Delay curves for $y_1$ pre and post the optimisation.

Figure 4.13.: Area-Delay curves for $y_2$ pre and post the optimisation.
Figure 4.14.: Area-Delay curves for $y_3$ pre and post the optimisation.

Figure 4.15.: Area-Delay curves for $y_4$ pre and post the optimisation.
These figures show that as the complexity of the polynomial grows, in this case, the maximum area reduction percentage decreases as follows: 55%, 44%, 36% and 17% respectively. This particular optimisation more than halves the area of an optional negating multiplication, but whose efficacy diminishes for larger polynomials.

To explore the benefit of the entire approach to optional negation found within Section 4.4 consider the following design where the only difference between the $p_i$ is the sign of the addends:

$$y_5 = \begin{cases} 
(s == 0) & ab + cd + ef + gh \\
(s == 1) & ab + cd + ef - gh \\
(s == 2) & ab + cd - ef + gh \\
(s == 3) & ab + cd - ef - gh \\
(s == 4) & ab - cd + ef + gh \\
(s == 5) & ab - cd + ef - gh \\
(s == 6) & ab - cd - ef + gh \\
(s == 7) & ab - cd - ef - gh \\
(s == 8) & -ab + cd + ef + gh \\
(s == 9) & -ab + cd + ef - gh \\
(s == 10) & -ab + cd - ef + gh \\
(s == 11) & -ab + cd - ef - gh \\
(s == 12) & -ab - cd + ef + gh \\
(s == 13) & -ab - cd + ef - gh \\
(s == 14) & -ab - cd - ef + gh \\
- & ab - cd - ef - gh 
\end{cases}$$

These polynomials only differ in their signs and thus optimising this design should demonstrate the sole benefits of the optional negation procedure. Applying the technique put forward in Section 4.4 results in the following design:

$$(ab + (c \oplus (s_2 \oplus s_3) + (s_2 \oplus s_3)))d + (e \oplus (s_1 \oplus s_3) + (s_1 \oplus s_3))f + (g \oplus (s_0 \oplus s_3) + (s_0 \oplus s_3))h - s_3 \oplus s_3$$
The area-delay curves for $y_5$ and its optimisation can be found in Figure 4.16. This particular optimisation exhibits an area improvement ranging between 37–76%. This technique seeks to fold any optional negations into a single polynomial with the intent that limited overhead is observed, the experiments shown here lends weight to the idea that this optimisation will always lead to an improvement in the hardware implementation costs.

![Area-Delay curves for $y_5$ pre and post the optimisation.](image)

Figure 4.16.: Area-Delay curves for $y_5$ pre and post the optimisation.

4.7.3. General Experiments

The previous two sections have provided point examples that stress elements of the overall flow. Elements of these experiments can be promoted to examples of the overall flow. In addition, the occurrence of mutually exclusive polynomials naturally occur in ALUs so we consider ALU inspired examples. The first will be that of standard ALU operations such as multiplication and multiply-add with the various possible negations. Higher order polynomials naturally occur in linear algebra applications where determinants are commonly calculated, a second example will be of computing
a mutually exclusive set of common determinants.

Section 4.7.1 provided insight into the efficacy of control logic minimisation. The three experiments performed in that section compared different selection logic strategies, the benefits of the full optimisation flow can be seen by comparing designs $y_{1,0}$ with $y_{1,2}$, $y_{2,0}$ with $y_{2,2}$ and $y_{3,0}$ with $y_{3,2}$. The area-delay curves for these comparisons can be found in Figures 4.17, 4.18 and 4.19 respectively.

Figure 4.17.: Area-Delay curves for $y_{1,0}$ and $y_{1,2}$. 
Figure 4.18.: Area-Delay curves for $y_{2,0}$ and $y_{2,2}$.

Figure 4.19.: Area-Delay curves for $y_{3,0}$ and $y_{3,2}$.
These benchmarks exhibit a maximum percentage area reduction of 47%, 77% and 79% respectively. The first of these results can be explained by noting that, essentially, design $y_{1,0}$ consists of only two different polynomials which design $y_{1,2}$ merges together. However, the other two designs consist of eight essentially different polynomials, which without performing the merging into a single polynomial along with careful control logic minimisation presented in this chapter results in this near five fold area increase.

The experiments presented in the previous Section 4.7.2 are examples of the full flow but are constructed such that the control logic minimisation is trivial. In addition to the experiments of the previous two sections which highlighted particular features of the flow, two additional examples have been considered. The following benchmark typifies an integer ALU when combined with modifiers for optional negation of inputs and outputs:

$$y_{6,0} = (s == 0)? \quad ab :$$
$$\quad (s == 1)? \quad -ab :$$
$$\quad (s == 2)? \quad c :$$
$$\quad (s == 3)? \quad -c :$$
$$\quad (s == 4)? \quad ab + c :$$
$$\quad (s == 5)? \quad ab - c :$$
$$\quad (s == 6)? \quad -ab + c : -ab - c$$

The following benchmark combines a set of typical determinants required for linear algebra operations including 3D determinant, triangle area and Vandermonde determinants:

$$y_{7,0} = (s == 0)? \quad \det \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} : (s == 1)? \quad \det \begin{vmatrix} 1 & 1 & 1 \\ a & b & c \\ d & e & f \end{vmatrix} :$$
$$\quad (s == 2)? \quad \det \begin{vmatrix} 1 & a & a^2 \\ 1 & b & b^2 \\ 1 & c & c^2 \end{vmatrix} :$$

The runtimes of the integer program when applying the optimisation flow to these benchmarks can be found Table 4.3 demonstrating that these take order of minutes to run. The number of variables in these integer programs
is the product of the number of mutually exclusive polynomials \( m \) and the maximum of the sum of the total degrees of the polynomials. The number of constraints is the product of the number of mutually exclusive polynomials \( m \) and the maximum of the number of terms in the polynomials. It is expected that typical applications of the procedure will have the number of variables and constraints bounded above by 100 and the *off-the-shelf* use of SMT solvers will be viable in the solution of the associated integer programs.

The results of applying the optimisations are \( y_{6,1} \) and \( y_{7,1} \) and are as follows:

\[
\begin{align*}
\tilde{x}_1 &= (s == 2) ? 0 : (s == 3) ? 0 : a \\
\tilde{x}_2 &= (s == 0) ? 0 : (s == 1) ? 0 : c \\
S_1[0] &= s_0 + s_1 - s_2 s_0 - s_2 s_1 s_0 \\
S_2[0] &= s_1 - s_1 s_0 + s_2 s_0 - s_2 s_1 s_0 \\
opnegx &= \tilde{x}_2 \oplus S_2 \\
y_{6,1} &= (b \tilde{x}_1 + \text{opnegx} + S_2 - S_1) \oplus S_1
\end{align*}
\]
\( \tilde{x}_1 = (s = 3) : c : 1 \quad \tilde{x}_2 = (s = 0) : g : (s = 1) : 1 : a \)

\( \tilde{x}_3 = (s = 0) : e : (s = 1) : e : c \quad \tilde{x}_4 = (s = 0) : g : b \)

\( \tilde{x}_5 = (s = 0) : b : (s = 3) : b : 1 \quad \tilde{x}_6 = (s = 0) : f : (s = 1) : f : a \)

\( \tilde{x}_7 = (s = 0) : 1 : (s = 1) : 1 : b \quad \tilde{x}_8 = (s = 0) : 1 : (s = 1) : 1 : c \)

\( \tilde{x}_9 = (s = 0) : d : (s = 1) : d : c \quad \tilde{x}_{10} = (s = 0) : i : (s = 3) : c : 1 \)

\( \tilde{x}_{11} = (s = 0) : f : (s = 1) : f : c \quad \tilde{x}_{12} = (s = 3) : a : 1 \)

\( \tilde{x}_{13} = (s = 0) : h : (s = 1) : 1 : a \quad \tilde{x}_{14} = (s = 0) : e : (s = 1) : e : b \)

\( \tilde{x}_{15} = (s = 0) : 1 : (s = 1) : 1 : a \quad \tilde{x}_{16} = (s = 0) : a : (s = 3) : a : 1 \)

\( \tilde{x}_{17} = (s = 0) : i : a \quad \tilde{x}_{18} = (s = 0) : d : (s = 1) : d : b \)

\( \tilde{x}_{19} = (s = 0) : 1 : (s = 1) : 1 : b \quad \tilde{x}_{20} = (s = 0) : h : (s = 3) : b : 1 \)

\( S_1 = s_1 \quad S_2 = 1 - s_1 \quad S_3 = s_1 \)

\( \text{opneg}\ x_2 = \tilde{x}_1 \oplus S_2 \)

\( \text{opneg}\ x_3 = \tilde{x}_4 \oplus S_3 \)

\( y_{7,1} = ((\text{opneg}\ x_2 + S_2) \tilde{x}_2 \tilde{x}_3 c + (\text{opneg}\ x_3 + S_3) \tilde{x}_5 \tilde{x}_6 \tilde{x}_7 - \tilde{x}_8 \tilde{x}_9 \tilde{x}_{10} b - \tilde{x}_{11} \tilde{x}_{12} \tilde{x}_{13} a + \tilde{x}_{14} \tilde{x}_{15} \tilde{x}_{16} \tilde{x}_{17} + \tilde{x}_{18} \tilde{x}_{19} \tilde{x}_{20} c - S_1) \oplus S_1 \)

The area-delay curves for these designs can be found in Figures 4.20 and 4.21.
Figure 4.20 shows an area benefit of 49-69% as well as a minimum delay 11% smaller for the optimised design. Figure 4.21 demonstrates a pronounced speed benefit with a 16% minimum delay improvement, a region where an area advantage of 16% can be observed and for large delays the result of the proposed flow actually produces an area degradation. A possible reason for this negative result is that the proposed flow has merged monomials of the form $abc$, $ab$, $ab^2$ and $ab^3$ into one monomial of the form $abcd$. However it is possible to implement each of the original monomials with a lower hardware implementation cost than $abcd$, due to the fact that simplified partial product arrays can be constructed for terms of the form $x^n$. For this reason, multiplicity within the monomials within the mutually exclusive polynomials it is required to implement, may render the proposed flow inappropriate. A summary of the percentage area benefits of the proposed flow for all of the experiments can be found in Table 4.4.

Although the general effectiveness of the proposed flow cannot be inferred from the few examples presented in this section, the results presented are encouraging. The control logic minimisation is optimal to the extent that it achieves the fewest multiplexors for the architecture considered. The use of the identity $(-1)^s x = (x - s) \oplus s$ in combination with the knowledge of the arithmetic SOP construction, achieves optional negation through the
Table 4.4.: Area Benefits of the Proposed Flow.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Maximum Area Reduction (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_5$</td>
<td>68</td>
</tr>
<tr>
<td>$y_{1.2}$</td>
<td>47</td>
</tr>
<tr>
<td>$y_{2.2}$</td>
<td>77</td>
</tr>
<tr>
<td>$y_{3.2}$</td>
<td>79</td>
</tr>
<tr>
<td>$y_1$</td>
<td>55</td>
</tr>
<tr>
<td>$y_2$</td>
<td>44</td>
</tr>
<tr>
<td>$y_3$</td>
<td>36</td>
</tr>
<tr>
<td>$y_4$</td>
<td>17</td>
</tr>
<tr>
<td>$y_5$</td>
<td>76</td>
</tr>
<tr>
<td>$y_{6.1}$</td>
<td>69</td>
</tr>
<tr>
<td>$y_{7.1}$</td>
<td>16</td>
</tr>
</tbody>
</table>

introduction of a single bit into the partial product array and a non critical XORing of the output. Thus it is possible to achieve optional negation with minimal hardware implementation cost impact. For these reasons, it is expected that the proposed flow will provide benefits in the general case.

4.7.4. Formal Verification Experiments

Formal verifications of all the designs used for the experiments was attempted, firstly using the word level equivalence checker [165] with and without the flow described in Section 4.6. The results can be found in Table 4.5 which were run on a 1.86 GHz Intel Xeon® machine with 4 GB of memory running Linux. These results demonstrate that the proposed technique can render intractable problems tractable and improve runtimes by up to 95%.
Table 4.5.: Formal Verification Runtimes.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Standard Runtime (s)</th>
<th>Runtime with Proposed Flow (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_5$</td>
<td>inconclusive after 24 hours</td>
<td>2</td>
</tr>
<tr>
<td>$y_{1,2}$</td>
<td>inconclusive after 24 hours</td>
<td>5</td>
</tr>
<tr>
<td>$y_{2,2}$</td>
<td>inconclusive after 24 hours</td>
<td>11</td>
</tr>
<tr>
<td>$y_{3,2}$</td>
<td>inconclusive after 24 hours</td>
<td>11</td>
</tr>
<tr>
<td>$y_1$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$y_2$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$y_3$</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$y_4$</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>$y_5$</td>
<td>inconclusive after 24 hours</td>
<td>2884</td>
</tr>
<tr>
<td>$y_{6,1}$</td>
<td>374</td>
<td>18</td>
</tr>
<tr>
<td>$y_{7,1}$</td>
<td>inconclusive after 24 hours</td>
<td>105</td>
</tr>
</tbody>
</table>

In conclusion, this chapter has presented a technique for the optimisation of the implementation of a set of mutually exclusive polynomials with integer coefficients as well as a method for proving formal correctness. Experiments have shown that implementing the optimisation requires minutes of runtime for the required integer program, consistent and considerable area reduction benefits of up to 79% and formal verifications which, once intractable, are now provable within seconds.
5. Lossy Fixed-Point Components

Once it is impractical for the precision of components to grow to accommodate the full accuracy of calculations, a certain amount of precision and accuracy must be lost in intermediate calculations. This is achieved by some type of rounding. If the infinitely precise answer $x$ is representable in the new precision then rounding should return this correct answer. Otherwise $x$ lies between two closest representable values, say $y_1$ and $y_2$:

$$y_1 < x < y_2$$

Typical rounding modes include:

- Round Towards Zero (RTZ) — $x$ is rounded towards whichever of $y_1$ or $y_2$ is closer to zero. For example, if $x > 0$, fixed-point and rounding was to whole integers then $x$ would be rounded to $\lfloor x \rfloor$.

- Round To Nearest, Ties to Even (RTE) — this rounding mode rounds to the nearer of $y_1$ and $y_2$. If $x$ is exactly halfway between $y_1$ and $y_2$, rounding is to the value which is even.

- Faithful Rounding (FR) — this rounding mode permits the rounding of $x$ to return either $y_1$ or $y_2$. This freedom means that functionally different designs may all be faithfully rounded. For example, RTZ and RTE can be viewed as examples of valid faithful rounding schemes.

Implementing these rounding modes will come with different implementation costs. Given that RTZ and RTE are examples of faithful rounding, an implementation of a faithful rounding scheme should be no worse in terms of cost than RTZ or RTE. In fact, the freedom provided by faithful rounding can provide significant gains in hardware implementation costs. Correct rounding, such as RTE and RTZ is not always necessary and in such cases faithful rounding can provide benefits. This chapter describes
the creation of faithfully rounded or lossy components which could be used in the implementation of fixed-point polynomials with rational coefficients. The components of interest are:

- Faithfully Rounded Integer Multiplication
- Faithfully Rounded Multioperand Addition
- Faithfully Rounded Constant Division

The architectures for all of these will be fully parallel ones, involving the summation of some partial product array. Consider designing a faithfully rounded integer multiplier with two 32 bit inputs, if it is not known a priori whether the design is faithfully rounded or not, it will require testing. This would mean $2^{64}$ simulations, which is infeasible. (There has even been research into how to perform the exhaustive simulation of faithfully rounded multipliers efficiently [177].) Given that this is just one instance of the architecture, testing all the potential vectors for all the multiplier sizes that users may want is completely infeasible. Thus faithfully rounded components, if they are to be used reliably, must be correct by construction and ready to be used as an off-the-shelf component. To enable industry adoption of these components, they must fit into a standard synthesis flow, to this end the architectures need to be directly embeddable in HDL and appropriate to then be synthesised with industry standard logic synthesis tools. Having made a faithfully rounded component, a natural question is whether or not the architecture can be maintained but the internal parameters tweaked to further reduce implementation cost (for example, removing more partial product bits from the array while still maintaining faithful rounding). The components that follow answer this question by providing conditions for faithful rounding which are not only sufficient but also necessary. Thus these components are not only faithfully rounded but moreover, given their architecture, there are no better components. Thus the sections that follow are necessarily entirely analytic and error bounds presented are proven to be tight. This chapter continues with Section 5.1 on integer multiplication, Section 5.2 on multioperand addition and Section 5.3 on constant division and concludes with a method by which these lossy components can be independently formally verified in Section 5.4.
5.1. Integer Multiplication

Lossy multipliers can be constructed by one of a variety of truncation schemes, this is where a number of partial product bits are removed from the multiplier array and some form of compensation is then performed [145], [134], [97], [90], [146] and [136]. In this section, truncation schemes are first introduced and then how guaranteed faithfully rounded multipliers can be constructed. Finally, to demonstrate the tightness of the error bound, the input vectors which give rise to the worse case error are shown.

5.1.1. Background on Truncation Multiplier Schemes

The structure of the majority of truncated multiplication schemes of two \( n \) by \( n \) bit inputs \( a \) and \( b \) producing an \( n \) bit output \( y \) is as follows. First, truncate the multiplier array by removing the value contained in the least significant \( k \) columns, denoted \( \text{val}(\Delta_k) \), prior to the addition of the partial products [145]. Second, a hardware-efficient function of the two multipli-cands \( f(a,b) \) is then introduced as compensation into column \( k \). Once the resultant array is summed, a further \( n - k \) columns are truncated, the result is then the approximation to the multiplication. The structure of the general multiplier truncation scheme is shown in Figure 5.1, the array in the figure is that of a traditional AND array multiplier. The underlying array may of course differ in structure, ranging from Booth arrays of various radix to squarer arrays and constant multiplication, etc [49]. Truncations of the AND array are first explored before exploring other array types.
The scheme may be summarised algebraically:

\[ y = 2^n \left\lfloor \frac{ab + 2^k f(a, b) - \text{val}(\triangle_k)}{2^n} \right\rfloor, \quad a, b, n, k \in \mathbb{Z}^+ \tag{5.1} \]

The error, compared to the precise answer, introduced by doing so is

\[ \varepsilon = ab - 2^n \left\lfloor \frac{ab + 2^k f(a, b) - \text{val}(\triangle_k)}{2^n} \right\rfloor \]

\[ \varepsilon = \left( (ab + 2^k f(a, b) - \text{val}(\triangle_k)) \mod 2^n \right) + \text{val}(\triangle_k) - 2^k f(a, b) \]

where \( T = (ab + 2^k f(a, b) - \text{val}(\triangle_k)) \mod 2^n \). A design that exhibits faithful rounding is one such that:

\[ \forall a, b \quad |\varepsilon| < 2^n \]

Note that if the correct answer is exactly representable, which occurs when the lower \( n \) bits of the multiplier result are all zero, then this perfect answer must be returned by a faithfully rounded scheme, otherwise \(|\varepsilon| \geq 2^n\).

Early truncation schemes considered \( f(a, b) \) being constant [145] and [95], referred to as Constant Correction Truncated schemes (CCT). Following these, the proposal to make \( f(a, b) \) a function of \( a \) and \( b \) appeared, termed Variable Correction Truncation (VCT) where the most significant column that is truncated is used as the compensating value for \( f(a, b) \) [97]. A hybrid between CCT and column promoting VCT has been proposed which only uses some of the partial product bits of the promoted column, termed Hybrid Correction Truncation [156]. Arbitrary functions of the most significant truncated column have been considered along with their linearisation; one of these linearisations requires promoting all but the four most extreme partial products bits and adding a constant, called LMS truncation due to the fact it targets the least mean square error [135] and [136]. Forming approximations to the carries produced by the summation of \( \triangle_k \) has also been put forward, termed carry prediction [120].

For Booth arrays, typically radix-4, their truncation history followed a
similar path to that of the AND arrays, first following a CCT type truncation [92] and column promotion [91]. Exhaustive simulation of the truncated part of the Booth array was used to design compensation circuitry based upon the conditional expectation of the error [80], or in order to construct Karnaugh maps of the ideal correction [24]. Recent work has focused on purely analytic techniques for computing the expected errors [108], [23].

Truncated arrays also have been considered for squarers, radix-4 and 16 and Booth squarer arrays [64], [39], [25]. Truncated arrays that perform multiplication by a fixed constant have been considered in [96] and [137], the former requiring exhaustive simulation in order to establish the truncation scheme and the latter performing analytic calculations to establish the optimal linear compensation factor that minimises the mean square error.

In terms of applications, DSP has been the main focus area but they also appear in creating floating-point multipliers where a one unit in the last place (ulp) accuracy is permitted [178]. The evaluation of transcendental functions has also been considered, utilising truncated multipliers as well as truncated squarers [66].

In general, given the focus has been on DSP applications, second order statistics of the error have been important. New truncation schemes often require exhaustive simulation as part of their construction or their validation. In advanced compensation schemes such as [120], it is commented that it is difficult to know what kind of error is being generated and while exhaustive searches were conducted for \( n \leq 8 \), for sizes above this, the only option was to resort to random test vectors. In [159], finding the best compensation function requires searching a space exponential in \( n \) and is only feasible for \( n < 13 \). Further the schemes either find compensating functions heuristically or attempt to minimise the average absolute error or mean square error.

Research looking at the absolute maximum error is less common. In [26] bounds for a truncated Booth radix-4 array are created. Truncated multipliers have been designed to minimise second order error [65], and their maximum absolute error has been bounded. An explicit attempt to create faithfully rounded multipliers, constructed by truncating, deleting and rounding the multiplication during the array construction, reduction and final integer addition can be found in [101].

Leading synthesis tools are extremely efficient at performing the sum-
mation of an arbitrary number of summands by avoiding expensive carry-
propagations and using redundant representations such as carry-save [182].
The array reduction is context-driven, depending on the timing and area
constraints and standard cell libraries in use. Access to the array reduction
or carry-save redundant signals is not possible from within the HDL code.
Creating HDL code which explicitly states which compressor cells to use
(full-adders, 4-to-2 compressors, etc.) in order to gain access to the inter-
mediate redundant representation will lack the timing and context driven
reduction achievable by the synthesis tool and will thus produce lower qual-
ity results. For these reasons the approach of [101] is not considered as a
viable option, as it modifies the multiplier array reduction directly and re-
quires access to intermediate carry-save signals. The only tight error bound
held within the literature is for the LMS schemes [65] and [63]. The aim is
to construct a variety of faithfully rounded truncated multipliers for a range
of schemes found within the literature and to compare their synthesis prop-
erties. Truncated AND arrays are first considered, as these are invariably
commutative, this is then followed by a consideration of other array types.

**CCT, VCT and LMS Multiplier Truncation**

CCT uses a single constant $C$ as the compensating function $f(a, b)$, as first
put forward in [145], so in this case:

$$f_{CCT}(a, b) = C$$

Column promoting truncated multiplication (VCT) takes $f(a, b)$ to be the
most significant column of $\triangle_k$ (denoted $col_{k-1}$) as put forward in [97], [178]:

$$f_{VCT}(a, b) = C + col_{k-1}$$

The LMS scheme, as put forward in Section 8 of [135], promotes the interior
of $col_{k-1}$ into $col_k$ leaving the extreme four partial products bits and adding
a constant one into column $n - 1$. This can be represented algebraically
by noting that elements of $col_{k-1}$ are $a_0b_{k-1}, a_1b_{k-2}, ..., a_{k-2}b_1, a_{k-1}b_0$ as
follows:

\[ f_{LMS}(a, b) = 2^{n-k-1} + \sum_{i=2}^{k-3} a_i b_{k-1-i} \]
\[ + \frac{1}{2} (a_0 b_{k-1} + a_1 b_{k-2} + a_{k-2} b_1 + a_{k-1} b_0) \]

5.1.2. Constructing Faithfully Rounded Multipliers

The literature on the analytic error properties of truncated multipliers is limited with simulation typically being used to establish correctness. Producing code for a faithfully rounded multiplier, using a truncated multiplier architecture, with a parameterisable input bit width is beyond the state of the art. In this section we present the analytic necessary and sufficient conditions for the faithful rounding of the CCT, VCT and LMS multiplier truncation schemes. This enables us to present how multipliers can be constructed which are known, a priori, to be faithfully rounded.

Necessary and Sufficient Conditions for CCT Faithful Rounding

In the case of CCT the error is:

\[ \varepsilon_{CCT} = T + \text{val}(\triangle_k) - 2^k C \]

Bounding CCT Error

Now \( T \) is the result of the summation in columns \( n-1 \) down to \( k \), so its smallest value is 0 and its largest \( 2^n - 2^k \) hence there exists the bound
\[ 0 \leq T \leq 2^n - 2^k. \]

Now \( \triangle_k \) can be full of zeros when \( a_{k-1:0} = b_{k-1:0} = 0 \)
(where \( a_{k-1:0} \) denotes the bits of \( a \) in columns \( k-1 \) down to 0) and full of ones when \( a_{k-1:0} = b_{k-1:0} = 2^k - 1 \), hence:

\[ 0 \leq \text{val}(\triangle_k) \leq \sum_{i=0}^{k-1} (2^k - 2^i) = (k-1)2^k + 1 \]

So an initial bound on \( \varepsilon_{CCT} \) becomes:

\[ -C2^k \leq \varepsilon_{CCT} \leq 2^n - (C - k + 2)2^k + 1 \]
The important question is whether or not there exist values for $a$ and $b$ where $T$ and $\text{val}(\triangle_k)$ can simultaneously achieve their lower/upper bound. The next section proves that this is possible and, hence, that this initial bound is, in fact, tight.

**CCT Error Bounds are Attained**

The lower bound is achieved when $T = \text{val}(\triangle_k) = 0$. Consider the case when $a_{k:0} = 100...000$, then:

$$T = \left( \left( a_{n-1:k+1}2^{k+1} + 2^k \right) b + C2^k \right) \mod 2^n$$

$$T2^{-k} - C = (2a_{n-1:k+1} + 1) b \mod 2^{n-k}$$

Now $2a_{n-1:k+1} + 1$ is odd, hence coprime to $2^{n-k}$, hence, regardless of the value of $C$, values of $a$ and $b$ can always be found such that any given $T$ can be achieved when $\text{val}(\triangle_k)$ is minimal.

The upper bound is achieved when $T$ and $\text{val}(\triangle_k)$ are both maximal. In the case when $a_{k-1:0} = b_{k-1:0} = 2^k - 1$:

$$T = \left( \left( a_{n-1:k}2^k + 2^k - 1 \right) \left( b_{n-1:k}2^k + 2^k - 1 \right) \right)$$

$$+ C2^k - \max(\text{val}(\triangle_k)) \mod 2^n$$

$$T2^{-k} - a_{n-1:k} \left( 2^k - 1 \right) - C - 2^k + k + 1$$

$$= b_{n-1:k} \left( a_{n-1:k}2^k + 2^k - 1 \right) \mod 2^{n-k}$$

Now $a_{n-1:k}2^k + 2^k - 1$ is odd hence coprime to $2^{n-k}$ hence, regardless of the value of $C$, values of $a_{n-1:k}$ and $b_{n-1:k}$ can always be found such that any given $T$ can be achieved when $\text{val}(\triangle_k)$ is maximal.

**The CCT Theorem**

Given the error bounds are tight, the necessary and sufficient conditions for the CCT scheme to be faithfully rounded can be derived from equation (5.2). This allows us to present our CCT theorem:

**Theorem 5.1.1** The CCT Theorem

The necessary and sufficient condition for the CCT scheme to be faithfully...
rounded is:

$$|\varepsilon_{CCT}| < 2^n \iff 2^{n-k} > C > k - 2$$

**Necessary and Sufficient Conditions for VCT Faithful Rounding**

In the case of VCT the error is:

$$\varepsilon_{VCT} = T + \text{val}(\Delta_k) - 2^k \text{col}_{k-1} - 2^k C$$

Providing tight bounds for $$\mu = \text{val}(\Delta_k) - 2^k \text{col}_{k-1}$$ is dealt with first.

**Bounding Maximum VCT $$\mu$$ Error**

Exhaustive simulation for small $$k$$ show particular forms for $$\Delta_k$$ when $$\mu$$ is maximal. When $$k$$ is odd, *e.g.* 7 these forms are:

```
0 1 0 1 0 1 1 0 0 1 0 1 0 1 0 1
0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1
0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1
0 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1
0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1
0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1
```

When $$k$$ is even, *e.g.* 8 these forms are:

```
0 1 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

These simulations motivated our first VCT theorem:
Theorem 5.1.2 The VCT Maximal Theorem

\[ \mu = \text{val}(\triangle_k) - 2^k \text{col}_{k-1} \]

\( \iff \) \( \text{col}_{k-1} = 0 \) and \( \text{col}_{k-2} \) is alternating

Proof:

- \( \mu(a_0, b_{k-1}) = a_0(-2^{k-1}b_{k-1} + b_{k-2}) + \text{const} \). Maximising \( \mu \) over \( a_0 \) and \( b_{k-1} \) gives \( a_0 = 1, b_{k-1} = 0 \) and \( b_{k-2} > 0 \). A symmetrical argument gives rise to \( a_0 = b_0 = 1 \) and \( a_{k-1} = b_{k-1} = 0 \).

- \( \mu(a_j, b_{k-1-j}) = -2^{k-1}a_jb_{k-1-j} + 2^ja_jb_{k-2-j} + 2^{k-1-j}b_{k-1-j}a_{j-1} + \text{const} \). Maximising \( \mu \) over \( a_j \) and \( b_{k-1-j} \) given that \( a_0 = b_0 = 1 \) gives rise to \( a_j \neq b_{k-1-j} \) and hence \( \text{col}_{k-1} = 0 \).

- Consider the case when there are two adjacent zeroes in \( \text{col}_{k-2} \) so there is a location where:

  \[
  \begin{align*}
  a_{j-1}b_{k-j} &= 0 \quad 0 \\
  a_jb_{k-j-1} &= 0 \quad 0 
  \end{align*}
  \]

  Assuming that \( a_j \neq b_{k-1-j} \) for all \( j \) and, by symmetry, it may be assumed that \( a_{j-1} = 1 \). Solving the above equations means that \( a_{j-1} = 1 \) and \( b_{k-1-j} = 000 \). If \( a_{j-1} = 1 \) had been set to 01 and \( b_{k-1-j} = 000 \) then \( \mu \) would have been increased by:

  \[
  \begin{align*}
  2^{k-2} + 2^{k-1-j}a_{j-2}b_{k-3} - 2^jb_{k-j-3} = & 0 \\
  > 2^{k-2} + 2^{k-1-j}a_{j-2}b_{k-2} - 2^j(2^{k-j-2} - 1) = & 2^j + 2^{k-j-1}a_{j-2} > 0 
  \end{align*}
  \]

Hence when \( \mu \) is maximal, adjacent zeroes never appear in \( \text{col}_{k-2} \).

- If adjacent ones were to appear in \( \text{col}_{k-2} \) then there would be a one in column \( \text{col}_{k-1} \), which contradicts the fact that when \( \mu \) is maximal \( \text{col}_{k-1} = 0 \).

From these four observations it can be concluded that \( \text{col}_{k-1} = 0 \) and \( \text{col}_{k-2} \) is an alternating binary sequence when \( \mu \) is maximal. In fact these
two conditions heavily restrict $a$ and $b$. When $k$ is odd, these conditions imply:

$$a_{k-1:0} = \frac{2^{k-1} - 1}{3} \quad \text{and} \quad b_{k-1:0} = \frac{2^k + 1}{3}$$

When $k$ is even, the conditions imply:

$$a_{k-1:0} = b_{k-1:0} = \frac{2^k - 1}{3} \quad \text{or} \quad a_{k-1:0} = b_{k-1:0} = \frac{2^{k-1} + 1}{3}$$

After much arithmetic, from these cases the following tight upper bound on $\mu$ can be derived:

$$\mu \leq \frac{(3k - 2)2^{k-1} + (-1)^k}{9}$$

**Bounding Minimum VCT $\mu$ Error**

Exhaustive simulation for small $k$ shows particular forms for $\triangle_k$ when $\mu$ is minimal. When $k$ is odd, e.g. 7, these forms are:

1 0 1 0 1 0 1 1 1 1 0 1 0 1 1 1
0 0 0 0 0 0 1 0 1 0 1 1
1 0 1 0 1 0 0 0 0 0 1 0 1 1 1
0 0 0 0 1 0 1 1
1 0 1 0 0 0 0 1 0 1 1
0 0 1 1
1 1

When $k$ is even, e.g. 8 these forms are:

1 0 1 0 1 0 1 1 1 1 0 1 0 1 0 1 1
0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 1 0 0 0 0 0 0 1 0 1 1 1
0 0 0 0 0 1 0 1 1
1 0 1 1 0 0 0 0 1 1
0 0 0 1 0 1 1
1 1

1 1
These simulations motivated our second VCT theorem:

**The VCT Minimal Theorem**

**Theorem 5.1.3** *The VCT Minimal Theorem*

\[
\mu = \text{val}(\triangle_k) - 2^k \text{col}_{k-1}
\]

\[\text{is minimal} \iff \text{col}_{k-1}'s \text{ extremes are 1}
\]

\[\text{col}_{k-1}'s \text{ interior alternates}
\]

**Proof:**

- If \(a_j \neq b_{k-1-j}\) and \(a_j = 1\) then \(\mu\) can be decreased by \(2^{k-1-j} - b_{k-2-j} > 0\) by setting \(b_{k-1-j} = 1\), hence \(\mu\) is minimal implies \(a_j = b_{k-1-j}\) for all \(j\).

- If \(a_0 = b_{k-1} = 0\) then \(\mu\) can be decreased by \(2^{k-1} - b_{k-2} > 0\) by setting \(a_0 = 1\). Hence \(a_0 = b_0 = a_{k-1} = b_{k-1} = 1\) and \(\text{col}_{k-1}\) begins and ends with 1.

- So \(\mu\) can now be treated as only a function of \(a\).

Define:

\[
\gamma(z) = \frac{1}{2^{k-1}} \mu (a = x_0 x_1 \ldots x_{m-1} z_{p-1} \ldots z_1 z_0 y_{q-1} \ldots y_1 y_0)
\]

\[= f(z) + g(z)X + h(z)Y + \gamma(0)
\]

where \(X = x_{m-1} \ldots x_1 x_0 / 2^m < 1\)

\[Y = y_{q-1} \ldots y_1 y_0 / 2^q < 1
\]

It is now shown that various strings within \(a\) never occur:

- \(\gamma(010) - \gamma(000) = (X + Y) / 2 - 1 < 0\) so ”000” never occurs

- \(\gamma(101) - \gamma(111) = -(X + Y) < 0\) when \(k > 3\) so ”111” never occurs

- If ”1001” were feasible then \(\gamma(1001) \leq \gamma(1010), \gamma(0101), \gamma(1011), \gamma(1101),\)
this would imply

\[ 0 \leq 1 + X - 4Y \]
\[ 0 \leq 1 + Y - 4X \]
\[ 0 \leq -1 + X + 2Y \]
\[ 0 \leq -1 + Y + 2X \]

But the only solution to this is \( X = Y = 1/3 \) which is not possible given that \( X \) and \( Y \) are finite binary numbers. Hence ”1001” never occurs.

- If ”0110” were feasible then \( \gamma(0110) \leq \gamma(0101), \gamma(1010), \gamma(0010), \gamma(0100) \),

  this would imply

\[ 0 \leq -2 + 4Y - X \]
\[ 0 \leq -2 + 4X - Y \]
\[ 0 \leq 2 - 2X - Y \]
\[ 0 \leq 2 - 2Y - X \]

But the only solution to this is \( X = Y = 2/3 \) which is not possible given that \( X \) and \( Y \) are finite binary numbers. Hence ”0110” never occurs.

From these observations it can be concluded that the \((k - 1)\)th column begins and ends with 1 and that it has an alternating interior when \( \mu \) is minimal.

These two conditions heavily restrict \( a \) and \( b \). When \( k \) is odd, these conditions imply:

\[ a_{k-1:0} = b_{k-1:0} = \frac{2^{k+1} - 1}{3} \quad \text{or} \quad a_{k-1:0} = b_{k-1:0} = \frac{5 \times 2^{k-1} + 1}{3} \]

When \( k \) is even, \( a \) and \( b \) are unique (up to swapping):

\[ a_{k-1:0} = \frac{5 \times 2^{k-1} - 1}{3} \quad \text{and} \quad b_{k-1:0} = \frac{2^{k+1} + 1}{3} \]
After much arithmetic, from these cases the following tight lower bound on $\mu$ can be derived:

$$\mu \geq -\frac{(3k + 7)2^{k-1} + (-1)^k}{9}$$

So in summary:

$$\varepsilon_{VCT} = T + \mu - 2^k C$$

Inserting the bounds for $\mu$ and knowing that $0 \leq T \leq 2^n - 2^k$ means:

$$\varepsilon_{VCT} \geq -\frac{2^{k-1}(3k+18C+7)+(-1)^k}{9} \leq 2^n + \frac{2^{k-1}(3k-18C-20)+(-1)^k}{9}$$  \hspace{1cm} (5.3)$$

The important question is whether or not there exist values for $a$ and $b$ where $T$ and $\mu$ can simultaneously achieve their lower/upper bound. The next section proves that this is possible and hence that this initial bound is, in fact, tight.

**VCT Error Bounds are Attained**

The lower bound is achieved when $T$ and $val(\triangle_k)$ are both minimal. When $k$ is even, $\mu$ is minimised by setting $a_{k-1:0} = \frac{(5 \times 2^{k-1} - 1)}{3}$ and $b_{k-1:0} = \frac{(2^{k+1} + 1)}{3}$, in this case $T$ is:

$$T = \left( a_{n-1:k}2^k + \frac{5 \times 2^{k-1} - 1}{3} \right) \left( b_{n-1:k}2^k + \frac{2^{k+1} + 1}{3} \right)$$

$$+ C2^k - \min(\mu) \mod 2^n$$

$$T2^{-k} - C - \frac{5 \times 2^{k+1} + 3k + 8}{18} - \frac{a_{n-1:k}2^{k+1} + 1}{3} = \frac{b_{n-1:k}\left( a_{n-1:k}2^{k} + \frac{5 \times 2^{k-1} - 1}{3} \right)}{mod 2^{n-k}}$$

Now $a_{n-1:k}2^k + (5 \times 2^{k-1} - 1)/3$ is odd hence coprime to $2^{n-k}$ hence, regardless of the value of $C$, $a$ and $b$ can always be found such that any given $T$ can be achieved when $\mu$ is minimal and $k$ is even. Similarly when $k$ is odd $a_{k-1:0} = b_{k-1:0} = \frac{(2^{k+1} - 1)}{3}$ or $a_{k-1:0} = b_{k-1:0} = \frac{(5 \times 2^{k-1} + 1)}{3}$ and the argument proceeds in an identical manner. It can therefore be concluded
that regardless of the value of $C$ values of $a$ and $b$ can always be found such that any given $T$ can be achieved when $\mu$ is minimal.

The upper bound is achieved when $T$ and $val(\triangle_k)$ are both maximal. When $k$ is odd, these conditions imply $a_{k-1:0} = (2^{k-1} - 1)/3$ and $b_{k-1:0} = (2^k + 1)/3$, in this case $T$ is:

$$T = \left(\left( a_{n-1:k}2^k + \frac{2^{k-1} - 1}{3}\right) \left(b_{n-1:k}2^k + \frac{2^k + 1}{3}\right) + C2^k - \max(\mu) \right) \mod 2^n$$

$$T2^{-k} - C - \frac{2^k - 3k + 1}{18} - a_{n-1:k} \frac{2^k + 1}{3}$$

$$= b_{n-1:k} \left( a_{n-1:k}2^k + \frac{2^{k-1} - 1}{3}\right) \mod 2^{n-k}$$

Now $a_{n-1:k}2^k + (2^{k-1} - 1)/3$ is odd hence coprime to $2^{n-k}$ hence, regardless of the value of $C$, values of $a$ and $b$ can always be found such that any given $T$ can be achieved when $\mu$ is maximal and $k$ is odd. Similarly when $k$ is even, $a_{k-1:0} = b_{k-1:0} = (2^k - 1)/3$ or $a_{k-1:0} = b_{k-1:0} = (2^{k-1} + 1)/3$ and the argument proceeds in an identical manner. It can therefore be concluded that regardless of the value of $C$ values of $a$ and $b$ can always be found such that any given $T$ can be achieved when $\mu$ is maximal.

**The VCT Theorem**

Given the error bounds are tight, necessary and sufficient conditions for the VCT scheme to be faithfully rounded can be derived from equation (5.3) on page 150. This allows us to present our VCT theorem:

**Theorem 5.1.4 The VCT Theorem**

The necessary and sufficient condition for the VCT scheme to be faithfully rounded is:

$$|\varepsilon_{VCT}| < 2^n \iff 3 \times 2^{n-k+1} - k - 2 > 6C > k - 7$$

151
Necessary and Sufficient Conditions for LMS Faithful Rounding

Error bounds for the LMS scheme were first reported in [63]:

\[-2^{n-1} - \frac{1}{9} \left(2^{k-4} \left(24k - 19 + 3(-1)^k\right) - 3 + 4(-1)^k\right) \leq \varepsilon_{LMS} \leq \frac{2^{n-k-1}}{9} \left(2^k(3k + 1) + 8(-1)^k\right) \] (5.4)

As stated in [63], in absolute value, the most negative error dominates. From this condition the necessary and sufficient condition for faithful rounding of the LMS scheme can be derived:

**Theorem 5.1.5 The LMS Theorem**

*The necessary and sufficient condition for the LMS scheme to be faithfully rounded is (for \(k > 5\)):*

\[ |\varepsilon_{LMS}| < 2^n \iff 9 \times 2^{n-k+1} > 6k - 5 + (-1)^k \]

Proof: as the negative bound of equation (5.4) dominates over the positive, in terms of absolute value, then the faithful rounding condition becomes:

\[ 2^{n-1} + \frac{1}{9} \left(2^{k-4} \left(24k - 19 + 3(-1)^k\right) - 3 + 4(-1)^k\right) < 2^{n-1} \]

Rearranging this equation and considering the cases of \(k\) odd and even separately produces the following conditions:

- \(9 \times 2^{n-k+1} > 6k - 4 + \frac{1}{2^{k-2}}\)  \(k\) even
- \(9 \times 2^{n-k+1} > 6k - 5 + \frac{7}{2^{k-2}}\)  \(k\) odd

Noting the integer value of the left hand side of these equalities, these conditions can be combined together (assuming that \(k > 5\), which will hold for any reasonably sized schemes):

\[ 9 \times 2^{n-k+1} > 6k - 5 + (-1)^k \]
Constructing Faithfully Rounded Multipliers

Necessary and sufficient conditions for faithful rounding of the three truncation schemes have now been presented. The aim is to create the lowest cost faithfully rounded designs. The hardware cost heuristic that larger \( k \) values remove more partial product bits and are thus more efficient to implement is used. Varying \( C \) has extremely limited impact on hardware resources used, however as an heuristic it is assumed that a small Hamming weight and small numerical value is desirable, so let \( \text{minHamm}(a, b) \) return the number of smallest value within the integers with smallest Hamming weight which exist in the interval \([a, b]\). This can be computed as follows, where \( a \) and \( b \) are \( p \) bits in length:

\[
\begin{align*}
\text{Input} & \quad a, b \quad p \text{ bit unsigned numbers} \\
\text{Output} & \quad c \quad \text{minHamm}(a,b) \\
\text{begin} & \\
\quad c = 0 \\
\text{for} & \quad i = p - 1 \quad \text{down to} \quad 0 \quad \text{begin} \\
\quad \quad \text{if} \quad a[i] == b[i] \quad \text{then} \quad c[i] = a[i]; \quad a[i] = 0; \\
\quad \quad \text{else} \quad c += 2^\lceil\log_2 a[i]\rceil; \quad \text{break;} \\
\quad \text{endif} \\
\text{end} \quad \text{break;} \\
\text{end}
\end{align*}
\]

The following values for \( k \) and \( C \) thus guarantee faithful rounding while minimising hardware costs for the three truncation schemes CCT, VCT.
Table 5.1.: X values for VCT worst case error vectors.

<table>
<thead>
<tr>
<th>n</th>
<th>k_{CCT}</th>
<th>C_{CCT}</th>
<th>k_{VCT}</th>
<th>C_{VCT}</th>
<th>k_{LMS}</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td>8</td>
<td>10</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>12</td>
<td>13</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>20</td>
<td>16</td>
<td>15</td>
<td>17</td>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>24</td>
<td>19</td>
<td>18</td>
<td>21</td>
<td>4</td>
<td>21</td>
</tr>
<tr>
<td>28</td>
<td>23</td>
<td>24</td>
<td>24</td>
<td>4</td>
<td>24</td>
</tr>
<tr>
<td>32</td>
<td>27</td>
<td>26</td>
<td>28</td>
<td>4</td>
<td>28</td>
</tr>
</tbody>
</table>

and LMS:

\[
k_{CCT} = \max \left( k : \exists \ C \text{ s.t. } 2^{n-k} > C > k - 2 \right) = \max \left( k : 2^n > (k - 1)2^k \right)
\]

\[C_{CCT} = \min \text{Hamm}(k_{CCT} - 1, 2^{n-k_{CCT}} - 1)\]

\[k_{VCT} = \max \left( k : \exists C \text{ s.t. } 3 \times 2^{n-k+1} - k - 2 > 6C > k - 7 \right) = \max \left( k : 3 \times 2^n > k2^k \right)\]

\[C_{VCT} = \min \text{Hamm} \left( \left\lceil \frac{k}{6} \right\rceil - 1, \frac{3 \times 2^{n-k+1} - k - 3}{6} \right)\]

\[k_{LMS} = \max \left( k : 9 \times 2^{n-k+1} > 6k - 5 + (-1)^k \right)\]

Example values for a variety of values of \( n \) can be found in Table 5.1.

The equations for the \( k \) and \( C \) parameters are simple enough to be embedded directly into HDL and allow the creation of correct by construction faithfully rounded integer multipliers.

5.1.3. Error Properties of Truncated Multipliers

To further stress the point that the error bounds that have been calculated are tight for the CCT and VCT, the inputs which give rise to the worst error are presented in the following sections.

CCT Worst Case Error Vectors

Given the bounds on \( \varepsilon_{CCT} \) the error is positive when the scheme has largest absolute worst case error if \( C2^k < 2^n - (C - k)2^k + 1 \) which simplifies
to $2C - k + 1 < 2^{n-k}$. The error is largest positive when $val(\triangle k)$ is maximal and $T = 2^n - 2^k$; the previous section shows that once $a_{n-1:k}$ is chosen, $b_{n-1:k}$ is fixed; hence there are $2^{n-k}$ possible worst case error vectors. The error is largest negative when $val(\triangle) = T = 0$. The previous section shows how this can be achieved when $a_{k:0} = 100...000$ and $b_{n-1:k} = a_{n-1:k} + 1 \mod 2^n - k$.

Now let $x$ be the largest integer such that $2^x$ divides $C$, then this implies that $b_{x-1:0} = 0$ and the equation reduces to:

$$a_{n-x-1:k}b_{n-k-1:x} + C/2^x = 0 \mod 2^{n-k-x}$$

This leaves $n-1$ bits of the inputs unconstrained, hence there are $2^{n-1}$ such error vectors. We assumed $a_k = 1$ but $ab$ is divisible by $2^{k+x}$ so powers of $2$ can be distributed between $a$ and $b$. There are $k + x + 1$ ways of doing so, hence in total there are $(x + k + 1)2^{n-1}$ total error vectors. In summary, we can present our theorem regarding CCT error vectors:

**Theorem 5.1.6 CCT Error Vectors**

If $2C - k + 1 < 2^{n-k}$ then there are $2^{n-k}$ worst case CCT error vectors, specified by:

$$a_{k-1:0} = b_{k-1:0} = 2^k - 1$$
$$b_{n-1:k} = -p(2^k + C - k + a_{n-1:k}(2^k - 1)) \mod 2^{n-k}$$

where $p$ is an integer satisfying $pa = 1 \mod 2^{n-k}$. Hence there is a $2^{-n-k}$ probability of encountering such inputs in simulation, provided all input sequences are equally likely.

Otherwise there are $(k + x + 1)2^{n-1}$ worst case error vectors, where $x$ be the largest integer, such that $2^x$ divides $C$, for $m = 0, 1, 2...k + x$ specified by:

$$a_m = 1 \quad b_{k+x-m-1:0} = a_{m-1:0} = 0$$
$$a_{n-k-x+m-1:m}b_{n-m-1:k+x-m} + C/2^x = 0 \mod 2^{n-k-x}$$

Hence there is a $(k + x + 1)2^{n-1}$ probability of encountering such inputs in simulation in this case, provided all input sequences are equally likely.
So for example, if a 32 bit faithfully rounded multiplier was built as in Section 5.1.2 then of the $2^{64}$ possible input combinations, there are only 32 which exercise the worst case error. Therefore there is a probability of $2^{-59}$ of seeing the worst case error in random simulation.

**VCT Worst Case Error Vectors**

Given the bounds on $\varepsilon_{VCT}$ the error is positive when the scheme has largest worst case error in absolute value if

$$\frac{2^{k-1}(3k + 18C + 7) + (-1)^k}{9} < 2^n + \frac{2^{k-1}(3k - 18C - 20) + (-1)^k}{9}$$

This simplifies to

$$4C + 3 < 2^{n-k+1}$$

Note that equality is not possible given the integer nature of $n$, $k$ and $C$. The exact error vectors can be found by following through the proofs in the previous sections, maximising and minimising $\mu$ and $T$. Note that in every case there are precisely $2^{n-k+1}$ error vectors. In summary, we can present our theorem regarding VCT error vectors:

**Theorem 5.1.7 VCT Error Vectors**

There are $2^{n-k+1}$ worst case VCT vectors for any given $n$, $k$ and $C$. Table 5.2 defines two different sets of values for $a_{k-1:0}$ and $b_{k-1:0}$. Further, $a_{n-1:k}$ can take any value and $b_{n-1:k}$ is then required to be:

$$b_{n-1:k} = -p(C + X/18 + a_{n-1:k}b_{k-1:0}) \mod 2^{n-k}$$

where $p$ is an integer satisfying $pa = 1 \mod 2^{n-k}$ and $X$ is defined in Table 5.2. The probability of encountering these worst case errors in simulation is thus $2^{2-n-k-1}$, provided all input sequences are equally likely.

So for example, if a 32 bit faithfully rounded multiplier was built as in Section 5.1.2 then of the $2^{64}$ possible input combinations, there are only 32
which exercise the worst case error. Therefore there is a probability of \(2^{-59}\)
of seeing the worst case error in random simulation.

Other error properties such as error expectation and variance have been calculated for the LMS scheme, these can be found in [135] and [136]. We have calculated these for the CCT scheme, these calculations can be found in Appendix A.

The next section on multioperand addition ultimately leads to the discovery of new architectures for faithfully rounded multipliers, after which experimental results are presented comparing all of the multiplier schemes.

### 5.2. Multioperand Addition

The previous sections dealt with the particular arrays associated with multiplication. There is a range of multiplier arrays found throughout the literature; AND arrays, Booth arrays of various radices, MUX arrays as well merged arrays performing multiply-add or sum-of-products. It would be useful to be able to truncate an arbitrary array such that the result is faithfully rounded. Given an arbitrary array is being considered, exploitation of any \(a\) priori correlations found within the array cannot be used. We now adopt a strategy akin to a CCT scheme, consider a truncated arbitrary array as in Figure 5.2.

<table>
<thead>
<tr>
<th>(\frac{4C + 3}{2^n - k + 1})</th>
<th>(k) odd</th>
<th>(a_{k-1:0})</th>
<th>(b_{k-1:0})</th>
<th>(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>00101...10101</td>
<td>01010...01011</td>
<td>(2^k - 3k + 19)</td>
</tr>
<tr>
<td>✓</td>
<td>x</td>
<td>00101...01011</td>
<td>00101...10101</td>
<td>(2^k - 3k + 19)</td>
</tr>
<tr>
<td>✓</td>
<td>x</td>
<td>01010...10101</td>
<td>01010...10101</td>
<td>(2^{k+1} - 3k + 16)</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>10101...01011</td>
<td>01010...10101</td>
<td>(2^{k-1} - 3k + 22)</td>
</tr>
<tr>
<td>x</td>
<td>✓</td>
<td>10101...01011</td>
<td>10101...10101</td>
<td>(2^{k+3} + 3k - 1)</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>11010...01011</td>
<td>11010...01011</td>
<td>(2^{k-1}25 + 3k + 17)</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>10101...01011</td>
<td>11010...10101</td>
<td>(2^{k+1}5 + 3k + 8)</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>10101...01011</td>
<td>11010...10101</td>
<td>(2^{k+1}5 + 3k + 8)</td>
</tr>
</tbody>
</table>
It is assumed that each partial product bit can vary independently and take values 0 or 1. Some partial product bits are discarded, let this set of discarded bits be denoted $\triangle$ and compensated by a fixed additive constant $C$. The scheme’s return value $F'$ should be a faithful rounding of the true full summation $F$ when the least significant $n$ bits are ignored. Algebraically, $F'$ can be defined as:

$$F' = 2^n \left\lfloor \frac{F - \text{val}(\triangle) + C}{2^n} \right\rfloor$$

where $\text{val}(\triangle)$ is the value of all the elements in $\triangle$ while respecting their binary weight. The error introduced by performing this approximation is:

$$\varepsilon = F - F' = ((F - \text{val}(\triangle) + C) \mod 2^n) + \text{val}(\triangle) - C$$

This error can be bounded by noting the modulo term ranges between 0 and $2^n - 1$. Note that these bounds may not be tight due to lack of knowledge of the array:

$$-C \leq \varepsilon \leq 2^n - 1 + \text{val}(\triangle) - C$$

For the scheme to be faithfully rounded then:

$$|\varepsilon| < 2^n$$

$$\text{val}(\triangle) C + 1 \leq 2^n$$

Setting $C$ to its maximal possible value, $2^n - 1$, places the least restriction.
on $\triangle$. The goal is to minimise the cost of implementing the truncated array while maintaining faithful rounding, the heuristic that summing fewer partial product bits will result in the smallest implementation cost is used. Therefore it is wished to maximise the number of elements in $\triangle$, denote this by $|\triangle|$. The optimisation problem then becomes:

$$\begin{align*}
    \text{max} & \quad |\triangle| \\
    \text{s.t.} & \quad \text{val}(\triangle) < 2^n
\end{align*}$$

**5.2.1. Optimal Truncation Theorem**

To solve this optimisation problem, introduce the variables $h_i$, the height of the array in column $i$ and $l_i$, the number of bits truncated from column $i$; example values for Figure 5.2 are illustrated in Figure 5.3.

![Diagram](image)

Figure 5.3.: Illustration of $h_i$ of $l_i$.

Note that the optimisation places no ordering on the bits in each column, merely their number. The optimisation problem then becomes:

$$\begin{align*}
    \text{max} & \quad \sum_{i=0}^{n-1} l_i \\
    \text{s.t.} & \quad \sum_{i=0}^{n-1} l_i 2^i < 2^n \\
                  & \quad l_i \leq h_i
\end{align*}$$

Let $k$ be the largest number of least significant columns that could be truncated while maintaining faithful rounding. More precisely (using the notation $\max(k : \text{cond})$ which returns the largest value of $k$ which satisfies the
condition \( cond \): \[
    k = \max \left( k : \sum_{i=0}^{k-1} h_i 2^i < 2^n \right)
\]

As shall be seen, the answer to the optimisation problem is closely related to \( k \). Let \( l_i^{opt} \) be the optimal values of \( l_i \) which maximises the objective function. The following lemmas contribute to the solution of the optimisation problem.

5.2.2. Lemma 1: \( l_i^{opt} = h_i \) for \( i < k \)

Proceeding by contradiction:

- If \( l_i^{opt} = 0 \) for \( i \geq k \) and there exists \( j < k \) such that \( l_j^{opt} < h_j \) then by the definition of \( k \), \( l_j \) can be increased to \( h_j \) thus increasing the objective while not violating the constraint.

- If there exists \( i \geq k \) and \( l_i^{opt} > 0 \) and \( j < k \) with \( l_j^{opt} < h_j \) then \( l_i^{opt} \) can be decremented and \( l_j^{opt} \) incremented. The objective is unchanged and the constraint is still met as the left hand side of the constraint is reduced by \( 2^i - 2^j > 0 \).

We may conclude that if there exists a supposedly optimal set of values for \( l_i \) such that \( l_i^{opt} < h_i \) for some \( i < k \), then by repeated application of the second point, truncations in column \( k \) or above can be exchanged for truncations in the least significant \( k \) columns. If all the truncations occur in the least significant \( k \) columns then these can include all partial product bits of the \( k \) columns, by the definition of \( k \). Hence it may be assumed that optimal \( l_i \) values satisfy \( l_i^{opt} = h_i \) for \( i < k \).

Restating the optimisation problem as a consequence of this lemma:

\[
    \begin{align*}
    \text{max} & \quad \sum_{i=k}^{n-1} l_i \\
    \text{s.t.} & \quad \sum_{i=0}^{n-k-1} l_{k+i} 2^i < 2^{n-k} - \sum_{i=0}^{k-1} h_i 2^{i-k} \\
    & \quad l_i \leq h_i
    \end{align*}
\]
It is useful to note that by the definition of $k$:

$$\sum_{i=0}^{k-1} h_i 2^i < 2^n \leq \sum_{i=0}^k h_i 2^i$$

$$0 < 2^{n-k} - \sum_{i=0}^{k-1} h_i 2^{i-k} \leq h_k$$

So the optimisation problem can be qualified as follows:

$$\max \sum_{i=k}^{n-1} l_i$$

$$\text{s.t. } \sum_{i=0}^{n-k-1} l_{k+i} 2^i < 2^{n-k} - \sum_{i=0}^{k-1} h_i 2^{i-k} \leq h_k$$

$$l_i \leq h_i$$

5.2.3. Lemma 2: $l_{i}^{opt} = 0$ for $i > k$

Proceeding by contradiction: say there exists $j > k$ such that $l_{j}^{opt} > 0$ then that implies that the constraint term contains terms of the following form:

$$... + l_{j}^{opt} 2^{j-k} + l_{k}^{opt} < 2^{n-k} - \sum_{i=0}^{k-1} h_i 2^{i-k} \leq h_k$$

Making the following transformation $l_j \rightarrow l_j - 1$ and $l_k \rightarrow l_k + 2^j - k$ will strictly increase the objective function and the first constraint will still be met. However, does the new $l_k$ still satisfy $l_k \leq h_k$? The first constraint is bounded by $h_k$ hence it was already true that:

$$l_{j}^{opt} 2^{j-k} + l_{k}^{opt} < h_k$$

$$(l_{j}^{opt} - 1)2^{j-k} + (l_{k}^{opt} + 2^j - k) < h_k$$

$$l_{k}^{opt} + 2^j - k < h_k$$

Hence the transformation still results in a feasible $l_k$. We may conclude that optimal values for $l_i$ when $i > k$ are all zero. This lemma shows that if there is a set of supposedly optimal values for $l_i$ which have truncations in a column above $k$ then these can be exchanged for more truncations in column $k$. 

161
Restating the optimisation problem as a result of this lemma:

\[
\begin{align*}
&\text{max } l_k \\
&\text{s.t. } l_k < 2^{n-k} - \sum_{i=0}^{k-1} h_i 2^{i-k} \leq h_k
\end{align*}
\]

The solution is thus:

\[
l_{k}^{opt} = \left\lceil 2^{n-k} - 1 - \sum_{i=0}^{k-1} h_i 2^{i-k} \right\rceil
\]

5.2.4. Faithfully Rounded Array Theorem

The result of the optimisation problem can now be stated as our faithfully rounded array theorem:

**Theorem 5.2.1** *Faithfully Rounded Array Theorem*

The optimal truncations \( l_i \) for an array with heights \( h_i \) returning a faithfully rounded result to the \( n \)th column are:

\[
l_{i}^{opt} = \begin{cases} 
 h_i & i < k \\
 2^{n-k} - 1 - \sum_{j=0}^{k-1} h_j 2^{j-k} & i = k \\
 0 & i > k 
\end{cases}
\]

where \( k = \max \left( k : \sum_{j=0}^{k-1} h_j 2^j < 2^n \right) \)

Given the uneven truncation of the optimal form, truncations performed using this method is termed *ragged*. Taking Figure 5.3 as an example, this situation has \( n = 8 \) and array heights for the least significant 8 columns \( \{9, 9, 9, 9, 7, 8, 5, 5\} \). Computing \( k \) gives 5 and \( l_5 = 0 \). The optimal truncations can be seen in Figure 5.4. Recall that the additive constant is always \( 2^n - 1 \), therefore a 1 needs to be added to every column (it is not added to the least significant \( k \) columns as its addition will have no impact).
5.2.5. Application to Multiplier Architectures

**Ragged Truncated Multipliers — AND Array (RAT)**

Applying this technique to a traditional AND array multipliers, in the case of the multiplication of two unsigned $n$ bit numbers with a faithfully rounded $n$ bit output; the array height of the $i$th column in the least significant $n$ columns is $i + 1$. Applying the Faithfully Rounded Array Theorem:

$$ l_{i}^{\text{opt}} = \begin{cases} 
  i + 1 & i < k \\
  2^{n-k} - 1 - \sum_{j=0}^{k-1} (j+1)2^{i-k} & i = k \\
  0 & i > k 
\end{cases} $$

where $k = \max \left( k : \sum_{j=0}^{k-1} h_j 2^j < 2^n \right)$

Simplifying produces:

$$ l_{i}^{\text{opt}} = \begin{cases} 
  i + 1 & i < k \\
  2^{n-k} - k & i = k \\
  0 & i > k 
\end{cases} $$

where $k = \max \left( k : (k-1)2^k < 2^n \right)$

As an example consider $n = 12$, then $k = 8$ and $l_8 = 8$, as illustrated in Figure 5.5.
Note that the truncations into column $k$ can be chosen such that the resultant truncated multiplier is still commutative.

**Ragged Truncated Multipliers — Booth Array (RBT)**

In the case of a Booth radix-4 array multipliers. For the multiplication of two unsigned $n$ bit numbers with a faithfully rounded $n$ bit output, the least significant $n$ columns of the multiplier array take the form as in Figure 5.6.

Given the specific structure of the array, the maximal value of any least significant $k$ columns of the array can be calculated as:

$$
\sum_{i=0}^{k-1} h_i 2^i = \left\lfloor \frac{k + 1}{2} \right\rfloor 2^k
$$
Now applying the Faithfully Rounded Array Theorem:

\[
l_i^{\text{opt}} = \begin{cases} 
    h_i & i < k \\
    2^{n-k} - 1 - \left\lceil \frac{k+1}{2} \right\rceil & i = k \\
    0 & i > k 
\end{cases}
\]

where \(k = \max\left(k : \left\lfloor \frac{k+1}{2} \right\rfloor 2^k < 2^n\right)\)

Simplifying produces:

\[
l_i^{\text{opt}} = \begin{cases} 
    h_i & i < k \\
    2^{n-k} - 1 - \left\lfloor \frac{k+1}{2} \right\rfloor & i = k \\
    0 & i > k 
\end{cases}
\]

where \(k = \max\left(k : (k+1)2^k < 2^{n+1}\right)\)

As an example the array in Figure 5.6 can be truncated, where \(n = 24\), in which case \(k = 20\) and \(l_{20} = 5\). The resultant truncation is illustrated in Figure 5.7.

Note that truncated Booth multipliers are non commutative.

**Faithfully Rounded RAT and RBT Multipliers**

Summarising the ragged truncation schemes; the parameters of the truncation scheme are \(k\), the number of least significant columns to remove, \(C\), the constant added into the array and \(l\), the number of bits to remove from
column \( k \). The parameters that describe the RAT and RBT schemes are:

\[
\begin{align*}
  k_{RAT} & = \max \left( k : 2^n > (k - 1)2^k \right) \\
  k_{RBT} & = \max \left( k : 2^{n+1} > (k + 1)2^k \right) \\
  C_{RAT} & = 2^n - 2^{k_{RAT}} \\
  C_{RBT} & = 2^n - 2^{k_{RBT}} \\
  l_{RAT} & = 2^{n-k_{RAT}} - k_{RAT} \\
  l_{RBT} & = 2^{n-k_{RBT}} - 1 - \left\lfloor \frac{k_{RBT} + 1}{2} \right\rfloor
\end{align*}
\]

5.2.6. Experiments

Five parameterisable pieces of HDL code were created for the CCT, VCT, LMS, RAT and RBT schemes which return a faithful rounding of an unsigned multiplication result as well as a reference multiplier which returns the RTE result, in order to see the benefit of truncation. Note that the construction of the schemes CCT, VCT and LMS as presented in the previous sections have the fewest partial product bits of that architecture which are faithfully rounded. This is due to the fact that their error bounds are tight. The RAT and RBT schemes do not necessarily have tight error bounds, but are of interest given the generality of their construction. There are schemes found within the literature whose error bounds are not tight, but can still be used to produce HDL which guarantees faithful rounding. These are included in the synthesis comparisons, there is a variant on VCT found in [134] and a CCT version of Booth radix-4 [92]. Note no comparison is performed against [101], [80], [24], [108] and [23] as their approach cannot be embedded into HDL, as they require offline compensation circuit construction or modifications to the synthesis process. Synthesis comparisons were performed for multipliers of size \( n = 16, 24 \) and 32. Synopsys Design Compiler was used to produce area-delay curves for the various multipliers. These experiments were performed for each value of \( n \), generating a range of delay and area points.

Before presenting the synthesis figures, inspecting the number of partial product bits in each scheme can provide some expectation in terms of implementation costs. Figure 5.8 contains the number of saved partial product bits over CCT for the VCT, LMS and RAT schemes for \( n = 8..32 \). Note that
for certain regions, particularly around $n = 16$ and $n = 32$, the RAT scheme has the fewest partial product bits and for $n = 24$ the LMS and VCT scheme have the minimal count. It can be shown analytically that $k_{\text{VCT}} \geq k_{\text{LMS}}$, so the VCT scheme will generally have no more partial product bits than LMS. RAT has been designed to minimise partial product bit count without reference to bit correlations, its error bounds may not be tight. In contrast, LMS and VCT have tight error bounds but a different architecture. Hence the partial product bit counts and synthesis will not strictly favour one architecture over another.

Figure 5.8.: Number of Saved Partial Product Bits over CCT.

Note that the values for $k$ are strictly less than $n - 1$, therefore the truncations are independent of the input bits $a_{n-1}$ and $b_{n-1}$ in the cases of the truncated AND arrays. So if $a$ and $b$ were two’s complement numbers then the analysis that gave rise to the necessary and sufficient conditions for faithful rounded would be unchanged. Therefore a standard AND array for two’s complement inputs can be truncated in an identical fashion to the unsigned multipliers presented here.

Turning to the actual synthesis results, truncated multipliers based upon an AND array are commutative and are compared against an AND array implementation of RTE in Figures 5.9, 5.10 and 5.11 (note the split $y$-axis). Truncated multipliers based upon a Booth radix-4 array are non-commutative and are compared against a Booth radix-4 array implementa-
tion of RTE in Figures 5.12, 5.13 and 5.14.

Figure 5.9.: Area/Delay Comparisons of Faithfully Rounded AND Array Multipliers $n=16$.

Figure 5.10.: Area/Delay Comparisons of Faithfully Rounded AND Array Multipliers $n=24$.  

168
Figure 5.11.: Area/Delay Comparisons of Faithfully Rounded AND Array Multipliers $n=32$.

Figure 5.12.: Area/Delay Comparisons of Faithfully Rounded Booth Array Multipliers $n=16$. 

169
These figures demonstrate that truncated AND array multipliers can provide an area benefit of 30-43% over the RTE multiplier, which increases as
As predicted from inspecting the partial product counts, the RAT scheme consistently exhibits the smallest area for $n = 16, 32$, whereas LMS and VCT dominates for $n = 24$. Truncated Booth arrays, in the form of the RBT design, offers a consistent improvement of 34-46% area compared to a Booth radix-4 RTE design. The RBT scheme is slightly superior to [92], due to the fact that [92] is CCT applied to a Booth array and RBT removes at least as many partial products as a CCT approach. It is interesting to note that within the set of non-Booth truncated multipliers none of the schemes has a strictly superior area for all bit widths, designers will have to choose based upon their particular hardware, accuracy and commutativity requirements.

These lossy components can thus be used off-the-shelf in any situation where faithful rounding can be tolerated. Further, the technique created for optimal truncation of an arbitrary array can be used for any SOP array or binary array reduction.
5.3. Constant Division

Finally, division by a constant is now addressed. Implementing integer division in hardware is expensive when compared to multiplication. In the case where the divisor is a constant, expensive integer division algorithms can be replaced by cheaper integer multiplications and additions. Creating bespoke hardware for integer division for a given invariant integer arises in a variety of situations, e.g. bespoke filtering, base conversions, certain caching algorithms, arithmetic with number formats where divisors of the form $2^n - 1$ are common. Previous work has mainly focused on software implementations given the lack of native integer division instructions within existing hardware [121]. Where hardware implementations are considered they are invariably sequential in nature. Certain divisors have been expanded into infinite products which translate into multiple shift and add instructions [109], [155]. An alternative proposed in [126], [72], [8], has been to replace the division by a single multiply-add instruction, computing $x/d \approx \lfloor \frac{ax+b}{2^k} \rfloor$ for suitable values of $a$, $b$ and $k$ (note that division by $2^k$ is zero cost).

The results in [141] show how only $n$ bit multiply-add operations are required and are thus optimal from a software perspective. However the work in [141] suffices with an $n$ by $n$ bit multiplication as opposed to finding the smallest bit widths as would be required by a hardware implementation. Invariant integer division using a serial multiplier is developed in [86]. Here, optimised hardware for computing invariant integer division is desired. Parallel multipliers and adders are appropriate for low latency hardware construction. The stated problem is to find integers $a$, $b$ and $k$ such that:

$$\text{Round} \left( \frac{x}{d} \right) = \left\lfloor \frac{ax+b}{2^k} \right\rfloor, \quad x \in [0,2^n-1]$$

The right hand side is easily realisable in hardware, as it is $(ax + b) \gg k$ which requires a simple multiply-add scheme and ignoring $k$ least significant bits of the output. Here, $x$ is assumed to be an unsigned $n$ bit number and it is assumed that $d$ is odd. Both of these assumptions can easily be relaxed, but are chosen here for ease of exposition. The case where $x$ is two’s complement is addressed in a subsequent section.
loss of generality, it can be assumed that $a$ is odd, because if it is even, then:

$$\left\lfloor \frac{ax + b}{2^k} \right\rfloor = \left\lfloor \frac{(a/2)x + (b/2)}{2^{k-1}} \right\rfloor = \left\lfloor \frac{(a/2)x + \lfloor b/2 \rfloor}{2^{k-1}} \right\rfloor$$

Which requires essentially exactly the same hardware to implement. Given $n$ and $d$, which values of $a, b$ and $k$ give rise to the various rounding modes RTZ, RTE and faithful? The subsequent sections answer this question, as well as finding the best choice of $a, b$ and $k$.

### 5.3.1. Conditions for Round Towards Zero Schemes

In order to realise the multiply-add scheme, it is required to find the conditions on $a, b$ and $k$ such that equation (5.5) is satisfied for all $x$. What follows is an analytic argument that produces these conditions. In the case of RTZ rounding, the functional requirement can be expressed as follows:

$$\left\lfloor \frac{x}{d} \right\rfloor = \left\lfloor \frac{ax + b}{2^k} \right\rfloor \quad \forall x \in [0, 2^n - 1]$$

$$0 \leq \frac{ax + b}{2^k} - \left( \frac{x - (x \mod d)}{d} \right) < 1$$

$$x \left( 1 - \frac{ad}{2^k} \right) - \frac{bd}{2^k} \leq x \mod d < x \left( 1 - \frac{ad}{2^k} \right) - \frac{bd}{2^k} + d \quad (5.6)$$

Thus it is required that $x \mod d$ (a sawtooth function in $x$) be bounded above and below by two lines of equal slope. Figure 5.15 shows an example for $d = 11$ and $n = 6$. In this example:

$$\frac{3x - 176}{256} \leq x \mod 11 < \frac{3x + 2640}{256} \quad x \in [0, 63]$$

Thus $$\left\lfloor \frac{x}{11} \right\rfloor = \left\lfloor \frac{23x + 16}{2^5} \right\rfloor \quad x \in [0, 63]$$

173
and division by 11 has been replaced by multiplication by 23, addition of 16 and a (free) truncation. In general it is necessary and sufficient to check that the upper bound of equation (5.6) is met for the peaks of $x \mod d$ which occur at $x = md - 1$ where $0 < m \leq \lfloor 2^n/d \rfloor$:

$$d - 1 < (md - 1) \left(1 - \frac{ad}{2^k}\right) - \frac{bd}{2^k} + d$$

$$m(ad - 2^k) < a - b$$

and the lower bound of equation (5.6) is met by the troughs which occur at $x = md$ for $0 \leq m \leq \lfloor 2^n/d \rfloor$.

$$md \left(1 - \frac{ad}{2^k}\right) - \frac{bd}{2^k} \leq 0$$

$$m(2^k - ad) \leq b$$

Now given that $a$ and $d$ are odd then $ad - 2^k \neq 0$. Depending on the sign of $ad - 2^k$, different values of $m$ will stress these inequalities. It can be concluded that the necessary and sufficient condition for the design to be
round towards zero compliant is:

\[-\frac{b+1}{2^n/d} < ad - 2^k < a - b \quad \text{if } ad - 2^k < 0\]

\[ad - 2^k < \frac{a-b}{2^n/d} \quad \text{if } ad - 2^k > 0\]

5.3.2. Conditions for Round To Nearest, Even and Faithfully Rounded Schemes

In the case of round to nearest, even, special care for rounding must be taken when \(x/d\) takes half integer values, namely when \(x/d = A + 1/2\) for some integer \(A\). However, rewriting this as \(2(x - Ad) = d\) this only holds for even \(d\) and \(d\) is odd by assumption. Thus the half way cases do not occur and the problem reduces to:

\[\left\lfloor \frac{x}{d} + \frac{1}{2} \right\rfloor = \left\lfloor \frac{ax+b}{2^k} \right\rfloor\]

By following a similar argument to that found in Section 5.3.1 one can derive the following necessary and sufficient conditions for the design to be round to nearest, even:

if \(ad - 2^k < 0\)

\[\frac{a(d-1) - 2b - 1}{2 \left( (2^{n+1} + d - 3)/2d \right) } < ad - 2^k < a \left( \frac{d + 1}{2} \right) - b\]

if \(ad - 2^k > 0\)

\[a \left( \frac{d-1}{2} \right) - b \leq ad - 2^k < \frac{a(d+1) - 2b}{2 \left( (2^{n+1} + d - 1)/2d \right) }\]

Faithful rounding requires that the exact answer be returned, if it is representable, otherwise it is permitted to return either of the two answers which are immediately above and below the true result. This means that for \(x = md\) where \(0 \leq m \leq \lfloor 2^n/d \rfloor\) the correct answer of \(m\) must be returned:

\[m = \left\lfloor \frac{amd + b}{2^k} \right\rfloor\]
Otherwise the constraint that either the answer above or below can be returned can be summarised as:

$$0 \leq \frac{ax + b}{2^k} - \left\lfloor \frac{x}{d} \right\rfloor < 2$$

Proceeding in a similar manner to Section 5.3.1 one can derive the following necessary and sufficient for the design to be faithfully rounded:

$$\left\lfloor \frac{2^n}{d} \right\rfloor (2^k - ad) \leq b < 2^k \quad \text{if } ad - 2^k < 0$$

$$\left\lfloor \frac{2^n}{d} \right\rfloor (ad - 2^k) < 2^k - b \quad \text{if } ad - 2^k > 0$$

5.3.3. Hardware Cost Heuristic

It is assumed that minimal hardware implementations will be formed when the number of partial product bits in $ax + b$ is minimal. This count is driven by the width of the partial product array which is $n + k$ in length. So it is chosen to minimise $k$ in the first instance. It turns out that having done this, there is no freedom in the value of $a$. However there is an interval in which $b$ may reside. In an effort to minimise the number of partial product bits, the set of valid $b$ values whose Hamming weight is minimal is found and then, of those, the one with smallest value will be chosen. This is working on the assumption that this limits the effect on the partial product array height which is normally highest in the middle of the array. As in the previous sections, let the function which finds this value for numbers in the interval $[a, b]$ be $\text{minHamm}(a, b)$.

5.3.4. Optimal Round Towards Zero Scheme

The hardware cost heuristic can now be applied to the necessary and sufficient conditions for a multiply-add scheme to perform RTZ rounding. Splitting this process on the whether $ad - 2^k$ is positive or negative:

**Optimal RTZ Scheme when $ad - 2^k > 0$**

In this case, it is required that:

$$ad - 2^k < \frac{a - b}{\left\lfloor \frac{2^n}{d} \right\rfloor}$$
Now note that the right hand side is strictly decreasing in $b$. So for any valid $a$, $b$ and $k$, $b$ can always be set to 0 and the condition will be met. Thus the condition reduces to:

$$(ad - 2^k) \left\lfloor \frac{2^n}{d} \right\rfloor < a$$

$$\frac{2^k}{d} < a < \frac{2^{k+1} \left\lfloor \frac{2^n}{d} \right\rfloor}{d} - 1$$  \hspace{1cm} (5.7)$$

Given that $a$ must be an integer, this provides a formula for $k_{opt}$, namely that of the smallest possible feasible value of $k$ such that:

$$k_{opt} = \min \left( k : \frac{1}{2^k} \left\lceil \frac{2^k}{d} \right\rceil < \left\lfloor \frac{2^n}{d} \right\rfloor - 1 \right)$$

So by the construction of $k_{opt}$, $a = \left\lfloor \frac{2^k}{d} \right\rfloor$ is valid as it satisfies equation (5.7). Now consider that $k_{opt}$ is the smallest valid $k$ hence:

$$\frac{1}{2^{k_{opt} - 1}} \left\lceil \frac{2^{k_{opt} - 1}}{d} \right\rceil \geq \left\lfloor \frac{2^n}{d} \right\rfloor - 1$$

$$\frac{2}{d} \left\lceil \frac{2^{k_{opt} - 1}}{d} \right\rceil \geq \frac{k_{opt} \left\lfloor \frac{2^n}{d} \right\rfloor}{d} - 1$$

$$\left\lceil \frac{2^{k_{opt}}}{d} \right\rceil + 1 \geq \frac{k_{opt} \left\lfloor \frac{2^n}{d} \right\rfloor}{d} - 1$$

Hence $a = \left\lfloor \frac{2^{k_{opt}}}{d} \right\rfloor + 1$ is invalid as it violates equation (5.7). It can be concluded that there is only one valid value for $a$ when $k = k_{opt}$. The design which minimises $k$ and satisfies $ad - 2^k > 0$ is unique and is defined by:

$$k_{opt}^+ = \min \left( k : \frac{2^k}{d} \left\lceil \frac{2^k}{d} \right\rceil > \left\lfloor \frac{2^n}{d} \right\rfloor - 1 \right)$$

$$a_{opt}^+ = \left\lceil \frac{2^{k_{opt}}}{d} \right\rceil$$

$$b_{opt}^+ = 0$$
Optimal RTZ Scheme when \( ad - 2^k < 0 \)

In this case, the following is required:

\[
- \frac{b + 1}{2^n/d} < ad - 2^k < a - b
\]

Hence \( b \) must necessarily reside in the following interval:

\[
b \in \left( (2^k - ad)\lfloor 2^n/d \rfloor, 2^k + a - ad - 1 \right]
\]

This interval must be non empty so:

\[
2^k + a - ad > (2^k - ad)\lfloor 2^n/d \rfloor
\]

\[
\frac{2^k}{d} > a > \frac{2^k((2^n/d) - 1)}{d\lfloor 2^n/d \rfloor - d + 1}
\]

Given that \( a \) must be an integer, this provides a formula for \( k_{opt} \):

\[
k_{opt} = \min\left( k : \frac{1}{2^k} \left\lfloor \frac{2^k}{d} \right\rfloor > \frac{\lfloor 2^n/d \rfloor - 1}{d\lfloor 2^n/d \rfloor - d + 1} \right)
\]

\[
\min\left( k : \frac{2^k}{2^k \mod d} > d \left\lfloor \frac{\lfloor 2^n/d \rfloor - 1}{d} \right\rfloor - d + 1 \right)
\]

In a similar manner to the previous section it can be shown that \( a = \lfloor 2^{k_{opt}}/d \rfloor \) is valid but \( a = \lfloor 2^{k_{opt}}/d \rfloor - 1 \) is not valid. The design which minimises \( k \) and satisfies \( ad - 2^k < 0 \) is unique in \( k \) and \( a \) and is defined by:

\[
k_{opt}^- = \min\left( k : \frac{2^k}{2^k \mod d} > d \left\lfloor \frac{\lfloor 2^n/d \rfloor - 1}{d} \right\rfloor - d + 1 \right)
\]

\[
a_{opt}^- = \left\lfloor \frac{2^{k_{opt}^-}}{d} \right\rfloor
\]

\[
b_{opt}^- = \min\text{Hammm}\left( (2^{k_{opt}^-} - a_{opt}^-d) \left\lfloor \frac{2^n}{d} \right\rfloor, 2^{k_{opt}^-} - a_{opt}^- (d - 1) - 1 \right)
\]

It can be shown that \( k_{opt}^+ \neq k_{opt}^- \), hence the design with smallest \( k \) is unique.
5.3.5. Optimal Rounding Schemes

Applying the techniques found in Section 5.3.4 to the other rounding schemes, results in the following condition:

\[(k, a, b) = (k^+ < k^-) \quad ? \quad (k^+, a^+, \text{minHamm}(Y^+(k^+, a^+))) \]
\[= (k^-, a^-, \text{minHamm}(Y^-(k^-, a^-)))\]

where

\[k^+ = \min \left( k : \frac{2^k}{(-2^k) \mod d} > X^+ \right)\]
\[k^- = \min \left( k : \frac{2^k}{2^k \mod d} > X^- \right)\]
\[a^+ = \left\lceil \frac{2^{k^+}}{d} \right\rceil \quad a^- = \left\lfloor \frac{2^{k^-}}{d} \right\rfloor\]

and the definition of \(X^\pm\) and \(Y^\pm\) can be found in Table 5.3.

| \(X^+\) | \(d\lfloor 2^n/d \rfloor - 1\) | \(2^n/d\) |
| \(X^-\) | \(d\lfloor 2^n/d \rfloor - d + 1\) | \(2^n/d\) |
| \(Y^+(k, a)\) | \((0, 0)\) | \((0, 0)\) |
| \(Y^-(k, a)\) | \((2^k - ad)\lfloor 2^n/d \rfloor, 2^k - a(d - 1) - 1\) | \((2^k - ad)\lfloor 2^n/d \rfloor, 2^k - 1\) |

5.3.6. Extension to Division of Two’s Complement Numbers

So far it has been assumed that \(x \geq 0\). Note that the three rounding modes considered so far are (or can be assumed in the case of FR to be) odd functions. So for \(x\) in some bounded negative interval one can find some \(a\)
and $b$ by proceeding as follows:

$$\text{Round} \left( \frac{x}{d} \right) = -\text{Round} \left( \frac{-x}{d} \right)$$

$$\quad = - \left\lfloor \frac{a(-x) + b}{2^k} \right\rfloor$$

$$\quad = (ax - b + 2^k - 1) >> k$$

Now because $\text{Round}(0/d) = 0$ then $b < 2^k$ and so $-b + 2^k - 1$ is in fact the bitwise inversion of the $k$ bits of $b$ namely $b[k-1:0]$. It can be concluded that if $x$ is an $n$ bit two’s complement number then:

$$\text{Round} \left( \frac{x}{d} \right) = (ax + b[k-1:0] \oplus x[n-1]) >> k$$

where $\oplus$ here is a bitwise XOR of the $k$ bits of $b$ and $k,a,b$ is constructed by the schemes presented to work for $x \in [0, 2^{n-1}]$.

5.3.7. Application to the Multiplication of Unsigned Normalised Numbers

An unsigned $n$ bit normalised number $x$ is interpreted as holding the value $x/(2^n - 1)$. Multiplication of these numbers thus involves computing the following:

$$\frac{y}{2^n - 1} \approx \frac{a}{2^n - 1} \frac{b}{2^n - 1}$$

$$y \approx \frac{ab}{2^n - 1}$$

The previously found results can be applied to the implementation of this design for the three rounding modes. In this case, $d = 2^n - 1$ and given that $ab \leq (2^n - 1)^2$ then $2^n - 1$ in the previous sections is now replaced by $(2^n - 1)^2$. Substituting these values into the previous sections gives rise to
implementations for the three rounding schemes: RTZ, RTE and FR:

\[
\begin{align*}
RTZ \left( \frac{ab}{2^n - 1} \right) &= \left\lfloor \frac{(2^n + 1)ab + 2^n}{2^{2n}} \right\rfloor \\
RTE \left( \frac{ab}{2^n - 1} \right) &= \left\lfloor \frac{(2^n + 1)(ab + 2^{n-1})}{2^{2n}} \right\rfloor \\
FR \left( \frac{ab}{2^n - 1} \right) &= \left\lfloor \frac{ab + 2^n - 1}{2^n} \right\rfloor
\end{align*}
\]

Note that the RTE case gives a generalisation and proof of the infamous Jim Blinn formula for such multiplication [11]. Note that the allowable interval for the additive constant in each case is \([2^n - 1, 2^n + 1], [2^n - 1(2^n + 1) - 2, 2^n - 1(2^n + 1)]\) and no freedom for the faithful rounding case.

5.3.8. Experiments

The equations that define the various rounding schemes are simple enough to be directly embedded in HDL. The implementation costs of RTZ, RTE, FR have been compared with the stated prior work in this area [141]. Synthesis experiments were performed for \(n = 16\) and \(d\) taking all odd values in the interval \([3,49]\) using Synopsys Design Compiler 2009.06-SP5 in ultra mode using the TSMC 65nm library Tcbn65lpc. Firstly, synthesis was performed which sought to minimise the logic delay and secondly minimise the area by setting the maximum delay of 2ns and allowing the tool to perform gate sizing. The results can be found in Figure 5.16. Dividing by a power of 2 utilises no hardware resource; note how the delay and area drops dramatically when \(d\) is around a power of 2 in these figures (note only odd values of \(d\) have been plotted). The faithfully rounded design exhibits up to a 20% speed improvement and 50% area improvement over the previous work [141]. The RTZ design, exhibits up to a 10% speed and 16% area improvement over the previous work, but due either to synthesis noise or an ill tuned hardware cost heuristic, can sometimes perform worse than the previous work.
In conclusion, components performing constant division with guaranteed rounding properties have been constructed which can be used in an off-the-shelf manner and support the input being unsigned or signed two’s complement.

5.4. Formal Verification of Lossy Components

If the components constructed in the previous sections are ever to be truly used off-the-shelf, in a manner similar to Synopsys DesignWare [163], they must be independently formally verified. The complication comes when attempting to formally verify faithfully rounded components, because of the freedom in the output. However, in the case of faithfully rounded unsigned $n$ bit multiplication and unsigned constant division, the rounded requirement means that:

\[
FR(ab) = \left\lfloor \frac{ab}{2^n} \right\rfloor \quad \text{or} \quad \left\lceil \frac{ab}{2^n} \right\rceil \\
FR\left(\frac{x}{d}\right) = \left\lfloor \frac{x}{d} \right\rfloor \quad \text{or} \quad \left\lceil \frac{x}{d} \right\rceil
\]
These conditions are properties which can be checked by a property checking tool such as Synopsys Hector [165]. The time taken to prove these properties for a faithfully rounded CCT multiplier for different values of $n$ and a faithfully rounded constant divisor with a 32 bit unsigned input for a variety of odd divisors can be found in Figures 5.17 and 5.18. The runtimes are of orders of minutes and seconds for the multiplier and divisions respectively, thus facilitating the creation of a fully verified library of lossy components in which a large number of instances of the parameterisable components can be verified.

![Figure 5.17.: Verification Runtimes for Faithfully Rounded CCT Multipliers of Size $n$.](image-url)
In conclusion, this chapter has presented architectures for a variety of integer multipliers which are guaranteed to be faithfully rounded and minimal (with respect to a hardware cost heuristic) which can be up to 46% smaller than their correctly rounded counterparts. An optimal method for truncating an arbitrary array has been presented and two new truncated multipliers discovered as a result. A parallel multiplier method for constant division has been put forward for RTZ, RTE and faithful rounding. The faithful rounding variant is up to 50% smaller than the correctly rounded designs. Finally an approach to verifying faithfully rounded components has been shown to verify these components in acceptable time and thus facilitate the creation of fully verified library of faithfully rounded components.
6. Lossy Fixed-Point Polynomial Synthesis

In the previous chapter, techniques were presented which exploited faithful rounding to minimise the number of partial product bits in integer multipliers, arbitrary arrays and constant division. These techniques can be augmented to exploit an arbitrary absolute error bound, thus producing a range of lossy components. If a fixed-point polynomial with rational coefficients is written as a combination of sum-of-products (this includes multipliers, squarers \((x^2)\), constant multipliers and sets of additions) and constant division then an implementation can use lossy versions of these operators to reduce implementation cost. The challenge is how best to distribute the allowable error between the operators to minimise implementation cost. This question has been explored in the context of word-length optimisation where bit widths of each intermediate variable has the potential to be independently varied [28]. The crucial observation and contribution of this chapter is that regardless of the heuristic used in navigation of the error landscape, the use of the lossy components will provide an improvement over the associated word level optimised equivalent.

Interestingly, word-length optimisation in circumstances where arbitrary arrays can be summed in parallel, such as in ASIC logic synthesis, can be highly detrimental. Consider these two designs:

\[
\begin{align*}
p_1 &= (ab + cd + ef + gh) >> n \\
p_2 &= ((ab) >> m) + ((cd) >> m) + ((ef) >> m) + ((gh) >> m)
\end{align*}
\]

Now \(p_1\) can be implemented by constructing a single partial product array consisting of all arrays for all the products and the entire array can be reduced as one, note that in doing so the value of any one of the products, \(e.g.\ ab\), is never produced in isolation. In the second case the construction of a
single array is not possible, an implementation of $p_2$ requires the separate calculation of products $ab$, $cd$, $ef$ and $gh$, each requiring array creation, reduction and an individual binary adder. The result of these four multiplications must be summed which requires reduction and another binary adder. Thus attempting to optimise intermediate bit widths can be prevent the use of efficient SOP implementations. Performing bit width optimisation at a sum-of-products level allows for the use of SOPs with their efficient implementation. In addition, raising the abstraction of operator to include SOPs reduces the number of nodes in the data flow graph (DFG) and thus reduces the number of degrees of freedom when allocating maximum allowable error to different parts of an implementation.

This chapter continues by first demonstrating how the techniques from the previous chapter can be augmented to exploit an arbitrary absolute error bound. This is followed by an exposition of the partial product arrays that could be required when implementing a general fixed-point polynomial in Section 6.2. This is then followed by the presentation of a novel technique for allocating error within a given DFG of SOPs and constant division operators in Section 6.3. Given the number of potential architectures and error allocations, a technique for determining the quality of a hardware implementation without the need for logic synthesis of the datapath is presented in Section 6.4. The chapter finishes with experimental evidence which compares the new technique to word-length optimisation, explores the quality of the error navigation approach and the heuristic hardware implementation cost function in Section 6.5.
6.1. Exploiting Errors for Arbitrary Arrays and Constant Division

The technique developed in Chapter 5 on page 157, gave the process by which the maximal number of partial product bits can be removed from an arbitrary binary array while maintaining a faithful rounding condition. This technique can be extended to deal with not just a faithful rounding condition that requires $|\varepsilon| < 2^n$ but an arbitrary fraction $u$ of $2^n$:

$$|\varepsilon| < u2^n$$

The argument previously presented is now restated in light of the introduction of parameter $u$. The error introduced by removing a set of the partial product bits $\triangle$ from the array, compensating with constant $C$, summing the resultant array and finally discarding the $n$ least significant columns is:

$$\varepsilon = \left((F - \text{val}(\triangle) + C) \mod 2^n\right) + \text{val}(\triangle) - C$$

where $F$ is the precise answer without any truncation and $\text{val}(\triangle)$ is the value of the set of bits within $\triangle$ with the binary weights of each columns being respected. This error can be bounded by noting that the modulo term ranges between 0 and $2^n - 1$.

$$-C \leq \varepsilon \leq 2^n - 1 + \text{val}(\triangle) - C$$

Note that these bounds may not be tight due to lack of knowledge of the array in question. If $|\varepsilon| < u2^n$ then this places the following restriction on $\triangle$ and $C$:

$$C < u2^n \quad 2^n - 1 + \text{val}(\triangle) - C < u2^n$$

$$\text{val}(\triangle) < (u - 1)2^n + C + 1 \leq (2u - 1)2^n$$

Setting $C$ to its maximal possible value, $\lceil u2^n - 1 \rceil$, places the least restriction on $\triangle$ and allows for the potential of removing the largest number of partial product bits. Having fixed $C$, the constraint on $\triangle$ is as follows:

$$\text{val}(\triangle) < (2u - 1)2^n$$

187
Note that as $u$ tends to $1/2$, $\triangle$ must be empty. This implies that a truncation scheme will never be correctly rounded. The rest of the previous argument is unaltered and the *Faithfully Rounded Array Theorem* on page 162 can be modified replacing $2^n$ with $(2u-1)2^n$. The result is our bounded error array theorem:

**Theorem 6.1.1 Bounded Error Array Theorem**

The optimal truncations $l_i$ for an array with heights $h_i$ returning a result with bounded error $u2^n$ are:

$$l_i^{opt} = \begin{cases} 
\left\lceil (2u - 1)2^{n-k} - 1 - \sum_{j=0}^{k-1} h_j2^{j-k} \right\rceil & i < k \\
(2u - 1)2^{n-k} - 1 & i = k \\
0 & i > k 
\end{cases}$$

where $k = \max \left( k : \sum_{j=0}^{k-1} h_j2^j < (2u - 1)2^n \right)$

Thus by removing $l_i^{opt}$ partial product bits from column $i$ in the array, introducing the constant $C = [u2^n - 1]$, summing the resultant array and discarding the least significant $n$ columns, will introduce an error which is strictly less than $u2^n$ in magnitude.

Now turning to constant division, the method for performing faithfully rounded constant division was presented in the previous chapter on page 179. This concerned the constant division of an unsigned $m$ bit number $x$ by $d$ and a multiply add implementation, more precisely:

$$\frac{x}{d} \approx (ax + b) >> k \quad x \in [0, 2^m - 1] \quad x, a, b, k \in \mathbb{Z}^+$$

The optimal values of $a$, $b$ and $k$ which minimises the partial product count of the implementation are as follows:

$$(a, b, k) = (k^+ < k^-)? (a^+, b^+, k^+) : (a^-, b^-, k^-)$$

$$k^\pm = \min \left( k : \frac{2^k}{(\mp 2^k) \mod d} > \left\lfloor \frac{2^m}{d} \right\rfloor \right)$$

$$a^+ = \left\lceil \frac{2^{k^+}}{d} \right\rceil \quad a^- = \left\lfloor \frac{2^{k^-}}{d} \right\rfloor$$

$$b^+ = 0 \quad b^- = \min_{\text{Ham}} \left( (2^{k^-} - a^- d) \lfloor 2^m/d \rfloor, 2^{k^-} - 1 \right) \quad (6.1)$$
where \( \text{minHamm}(x, y) \) returns the number of smallest value from the set of numbers of smallest Hamming weight within the integer interval \([x, y]\). In order to leverage this work note that if a faithful rounding \( f \) of \( x/(2^r d) \) is computed then:

\[
\left| f - \frac{x}{2^r d} \right| < 1 \\
\left| f2^r - \frac{x}{d} \right| < 2^r
\]

So if an error of \( e \) is permitted in a constant division then finding the largest \( r \) and associated faithful rounding \( f \) that satisfies the following will guarantee the required error is met:

\[
\left| f2^r - \frac{x}{d} \right| < 2^r \leq e
\]

So \( r = \lfloor \log_2(e) \rfloor \). In conclusion, if it is required to implement \( x/d \) for integer variable \( x \) and constant integer \( d \) with a maximum absolute error of strictly less than \( e \) then implement a faithful rounding of \( x/(2^r d) \) where \( r = \lfloor \log_2(e) \rfloor \), the result should then be interpreted has having \(-r\) fractional bits.

### 6.2. Array Constructions

The arrays that are required when dealing with a general SOP are multiplications, squarers \( a^2 \) and constant multiplications. The array construction for the common multiplier architecture Booth radix-4 has been discussed in the background material in Chapter 2 on page 40. A simple and efficient squarer array can be derived by considering using an AND array for \( a^2 \). Note how the following AND array for a squarer with a four bit input \( a \) can
be simplified:

\[
\begin{array}{ccccccccc}
  a_3 \land a_0 & a_2 \land a_0 & a_1 \land a_0 & a_0 \land a_0 \\
  a_3 \land a_1 & a_2 \land a_1 & a_1 \land a_1 & a_0 \land a_1 \\
  a_3 \land a_2 & a_2 \land a_2 & a_1 \land a_2 & a_0 \land a_2 \\
  a_3 \land a_3 & a_2 \land a_3 & a_1 \land a_3 & a_0 \land a_3 & + \\
  a_3 \land a_2 & a_3 \land a_0 & a_1 & a_0 & a_0 \\
  a_3 & \\
  a_2 & \\
\end{array}
\]

Such simplifications lead, in general, to an \( n \) bit squarer which has an array with \( n(n+1)/2 \) partial product bits.

Attention is now turned to the construction of an array that performs constant multiplication. One method for performing constant multiplication is via canonical signed digit representations [132]. This technique is a generalisation of the following observations when multiplying by numbers with particular binary strings:

\[
\begin{align*}
11111111_b x &= (x << 8) - x \\
11101111_b x &= (x << 8) - (x << 4) - x \\
110101011_b x &= (x << 9) - (x << 6) - (x << 4) - (x << 2) - x
\end{align*}
\]

These transformations can be viewed as first encoding the binary string in a canonical signed digit form as follows:

\[
\begin{align*}
1111111 &= 10000000T \\
11101111 &= 1000TT00T \\
110101011 &= 100TT0T0T0T \\
\end{align*}
\]

where \( T \) indicates -1 in a given binary location. This new representation is unique provided no adjacent bits are both non zero. The construction of this canonical signed digit form can be achieved by the following right to
left algorithm:

\textbf{Input} \ a[n - 1 : 0] \ binary \ string \ to \ be \ encoded
\textbf{Output} \ c[n : 0] \ equivalent \ canonical \ signed \ digit \ representation
\begin{verbatim}
begin
\quad a_{-1} = 0
\quad t_{-1} = 0
\quad for \ i = 0 \ to \ n \ do \ {
\quad \quad t_i = t_{i-1} \land (a_i \oplus a_{i-1})
\quad \quad c_i = t_i? (a_{i+1}? 1 : 0) : 0
\quad}
end
\end{verbatim}

The array is then formed by computing the canonical signed digit representation then using the non zero elements of the result to determine where the shifted copies of \( x \) and their negations should be placed. Note that negation can be achieved by inverting \( x \) and adding appropriate constant ones into the array.

These three array types, Booth, squarer and canonical signed digit allow the efficient creation of an array for any SOP. These arrays can be truncated, via the technique presented in the previous section, to exploit an arbitrary error freedom.
6.3. Navigation of the Error Landscape

The question now arises of how best to distribute the error amongst the SOPs and constant division units in order to minimise the implementation cost of a polynomial $p$. Figure 6.1 contains a schematic of an example data flow graph (DFG) consisting of SOPs and constant division operators.

![Figure 6.1.: Example SOP and Constant Division DFG.](image)

Each of these operators will (in the case of the constant division operators) or can (in the case of the SOPs) introduce an error. The greater the maximum absolute error, the greater the potential for hardware implementation cost reduction. Now word-length optimisation has been shown to be NP-hard in general [30] and thus heuristics have been developed to explore the error landscape. Recalling these techniques, which were discussed in the background material on page 49, one of these used differentiation of the polynomial to produce a notion of sensitivity of the output to a given internal precision change [133] and hardware heuristic cost functions to step through the space of possible precisions in [62] and [29]. The crucial observation is that regardless of the heuristic used in navigation of the error landscape, the use of the lossy components will provide an improvement over the associated word level optimised equivalent. Taking inspiration from [133], [62] and [29], the heuristic developed here exploits the fact that the design in question is polynomial and that partial product bit count determines the quality of the implementation.
6.3.1. Developing a Heuristic for the Navigation of the Error Landscape

In keeping with the approach taken in the previous chapter, the development of the heuristic is based upon removal of the greatest number of partial product bits. Ultimately the number and configuration of the partial products determine the characteristics of the cost of implementation. Let \( e_i \) be the maximum absolute error permitted in the \( i \)th operator and \( R_i(e) \) be the number of partial product bits removed in operator \( i \) due to exploiting the error freedom in all operators with bounds defined by the elements of \( e \). The total number of partial product bits removed is then:

\[
\sum_i R_i(e)
\]

Let \( \hat{p}(x, \varepsilon) \) be the actual value of the implementation which introduces error \( \varepsilon_i \) at node \( i \) (so \( |\varepsilon_i| \leq e_i \)), then the absolute error requirement is, for some user defined bound \( \eta \):

\[
|p(x) - \hat{p}(x, \varepsilon)| \leq \eta
\]

Expanding the left hand side with respect to \( \varepsilon \) means that for some coefficients \( c(x) \) this condition can be written as follows:

\[
\left| \sum_{\alpha} c_{\alpha}(x) \varepsilon^{\alpha} \right| \leq \eta
\]

A bound on the left hand side is:

\[
\left| \sum_{\alpha} c_{\alpha}(x) \varepsilon^{\alpha} \right| \leq \sum_{\alpha} \left( \max_x |c_{\alpha}(x)| \right) \varepsilon^{\alpha}
\]

So a sufficient, but potentially non necessary, condition for the scheme to meet the user defined absolute error requirement is:

\[
\sum_{\alpha} \lambda_{\alpha} e^{\alpha} \leq \eta
\]

where \( \lambda_{\alpha} = \max_x |c_{\alpha}(x)| \)
The following optimisation problem may then be posed, which maximises the number of partial product bits removed with respect to the maximum absolute errors introduced at each node:

$$\max \sum_i R_i(e)$$
subject to $$\sum_\alpha \lambda_\alpha e^\alpha \leq \eta$$

The maximum partial product removal will be achieved by fully utilising the error freedom, so the constraint can be set as equality:

$$\max \sum_i R_i(e)$$
subject to $$\sum_\alpha \lambda_\alpha e^\alpha = \eta$$

This can be solved by creating the Lagrangian using a Lagrange multiplier $T$:

$$L = \sum_i R_i(e) + T \left( \sum_\alpha \lambda_\alpha e^\alpha - \eta \right)$$

Turning points of $L$ are potential optimal solutions to the optimisation problem, taking partial derivatives with respect to $e_j$ gives rise to the following equation:

$$\sum_i \frac{\partial R_i}{\partial e_j} + T \sum_\alpha \lambda_\alpha \frac{\partial e^\alpha}{\partial e_j} = 0$$

So an extremum of the original optimisation occurs when:

$$\sum_i \frac{\partial R_i}{\partial e_j} \propto \sum_\alpha \lambda_\alpha \frac{\partial e^\alpha}{\partial e_j}$$

(6.2)

Now $\frac{\partial R_i}{\partial e_j}$ crucially depends on the current state of all of the values of $e$. These values can provide guidance in which direction to move $e$, but the starting state, like all heuristic schemes, is crucial. In the absence of any knowledge of $R_i$, assume that $e_j$ only affects $R_j$ and that $R_j$ is proportional to $\log e_j$ ($\log e_j$ will be proportional to the number of partial product columns that could be truncated when exploiting the error $e_j$). Under this
assumption the following holds:
\[
\frac{\partial R_j}{\partial e_i} = \begin{cases} 0 & \text{if } i \neq j \\ \frac{1}{e_j} & \text{if } i = j \end{cases}
\]

So a potential starting point for an iterative heuristic improvement strategy would be using values of \(e_i\) which satisfy the following equations for some constant \(T\):
\[
\sum_{\alpha} \lambda_{\alpha} e_{j} \frac{\partial e_{\alpha}}{\partial e_{j}} = T \quad \text{for all } j \\
\sum_{\alpha} \lambda_{\alpha} e^{\alpha} = \eta
\]

Now equation (6.2) can be used to iteratively improved via the following process:

- Solve for \(e_i\) for some constant \(T\):
  \[
  \sum_{\alpha} \lambda_{\alpha} e_{j} \frac{\partial e_{\alpha}}{\partial e_{j}} = T \quad \text{for all } j \\
  \sum_{\alpha} \lambda_{\alpha} e^{\alpha} = \eta
  \]

- Compute \(\frac{\partial R_i}{\partial e_j}\) for all \(i\) and \(j\).

- While the following are not equal for all \(j\):
  \[
  \left\{ \frac{1}{\sum_{i} \frac{\partial R_i}{\partial e_j}} \sum_{\alpha} \lambda_{\alpha} \frac{\partial e_{\alpha}}{\partial e_{j}} \right\}
  \]

  Replace \(e\) by \(f\) where \(f\) satisfies, for some constant \(T\):
  \[
  f_{j} \sum_{\alpha} \lambda_{\alpha} \frac{\partial f_{\alpha}}{\partial f_{j}} = T e_{j} \sum_{i} \frac{\partial R_i}{\partial e_{j}} \quad \text{for all } j \\
  \sum_{\alpha} \lambda_{\alpha} f_{\alpha} = \eta
  \]

This iterative process should move the variables \(e_i\) towards a point at which equation (6.2) is satisfied. This heuristic process relies on giving
values to $\frac{\partial R_i}{\partial e_i}$, the examples in the following sections illuminate this process and give rise to the presentation of a full procedure for navigating the error landscape.

6.3.2. Example Navigations of the Error Landscape

The first example is that of a cubic polynomial, which is used to given a meeting to $\frac{\partial R_j}{\partial e_i}$ in the case of SOPs. The second example is bilinear interpolation which requires constant division and is used to give a value to $\frac{\partial R_j}{\partial e_i}$ in this case. This first two examples have implementing polynomials $\hat{p}(x, \varepsilon)$ which are linear in variables $\varepsilon_i$, this simplifies the navigation considerably. The third example, an instance of a Gabor filter, has an implementing polynomial $\hat{p}(x, \varepsilon)$ which is non linear in the $\varepsilon_i$ variables. Together, these examples clarify the intent of the error navigating procedure.

Cubic Polynomial

Consider the following cubic polynomial:

$$at^3 + bt^2 + ct + d$$

With unsigned $n$ bit integer inputs $a, b, c$ and $d$ as well as a purely fractional $n$ bit variable $t$. Replacing $t$ by $t/2^n$ means that $t$ can now be interpreted as an $n$ bit unsigned integer:

$$a \left(\frac{t}{2^n}\right)^3 + b \left(\frac{t}{2^n}\right)^2 + c \left(\frac{t}{2^n}\right) + d$$

$$= \frac{at^3 + 2^n bt^2 + 2^{2n} ct + 2^{3n} d}{2^{3n}}$$

A natural question, is then how best to compute a faithful rounding of this expression, returning an integer. More precisely, defining the infinitely precise result $p$:

$$p = at^3 + 2^n bt^2 + 2^{2n} ct + 2^{3n} d$$
An implementation returning \( \hat{p} \) will be a faithful rounding of the original problem if, for all inputs \( a, b, c, d \) and \( t \):

\[
|p - \hat{p}| \leq 2^{3n} - 1
\]  

(6.4)

In this case, the globally acceptable error is \( \eta = 2^{3n} - 1 \). Consider a basic implementation of \( p \) which computes the powers of \( t \) first, as depicted in Figure 6.2.

![Figure 6.2.: Basic Implementation of Cubic Polynomial.](image)

Errors can be introduced into each of three operators depicted, say these errors are \( \varepsilon_1, \varepsilon_2 \) and \( \varepsilon_3 \) and that the value for \( p \) returned by such error introductions is \( \hat{p} \):

\[
\hat{p} = a((t^2 + \varepsilon_1)t + \varepsilon_2) + 2^n b(t^2 + \varepsilon_1) + 2^{2n} c t + 2^{3n} d + \varepsilon_1
\]

The faithful rounding condition in equation (6.4) then requires:

\[
|p - \hat{p}| \leq 2^{3n} - 1
\]

\[
|(at + 2^n b)\varepsilon_1 + a\varepsilon_2 + \varepsilon_3| \leq 2^{3n} - 1
\]

So in this case \( p - \hat{p} \) is linear in variables \( \varepsilon_i \). In such cases the heuristic iterative algorithm held within equations (6.3) reduces to the following process:

- Set \( e_i = \eta/(\lambda_i n) \) where \( n \) is the number of operators.
- Compute \( \frac{\partial R_i}{\partial e_j} \) for all \( i \) and \( j \).
• While the following are not equal for all $j$:

$$\left\{ \frac{\lambda_j}{\sum_i \frac{\partial R_i}{\partial e_j}} \right\}$$

Replace $e$ by $f$ where $f$ satisfies:

$$f_j = \frac{\eta e_j \sum_i \frac{\partial R_i}{\partial e_j}}{\lambda_j \sum_i e_j \frac{\partial R_i}{\partial e_j}} \quad (6.5)$$

In this case the $\lambda_i$ terms are as follows:

$$\lambda_1 = \max |at + 2^n b| = (2^n - 1)(2^{n+1} - 1)$$
$$\lambda_2 = \max |a| = 2^n - 1$$
$$\lambda_3 = 1$$

Using the heuristic navigation algorithm held within equations (6.5) requires the following starting values for $e_i$:

$$e_1 = \frac{\eta}{\lambda_1 n} = \frac{2^{3n} - 1}{3(2^n - 1)(2^{n+1} - 1)}$$
$$e_2 = \frac{\eta}{\lambda_2 n} = \frac{2^{3n} - 1}{3(2^{n+1} - 1)}$$
$$e_3 = \frac{\eta}{\lambda_3 n} = \frac{2^{3n} - 1}{3} \quad (6.6)$$

Now the techniques on truncating arbitrary arrays can be applied. Applying the truncation scheme with such an error bound will result in array truncations, a constant addition and finally the removal of the least significant $r$ columns. An implementation of the cubic without any rounding would be as follows:

$$t2[2n - 1 : 0] = t^2$$
$$t3[3n - 1 : 0] = t \times t2$$
$$p[4n - 1 : 0] = a \times t3 + ((b \times t2) << n) + ((c \times t) << 2n) + (d << 3n)$$

If each of these SOPs were truncated using the truncation of arbitrary array technique with a permissible maximum absolute error bound of $e_i = u_i 2^r$
where \( u_i \in (1/2, 1] \) then the implementation would have the following bit widths (denoting \( \text{trunc}(op, \text{error}) \) as the result of applying truncation to operator \( op \) by exploiting the error freedom \( \text{error} \)): 

\[
\begin{align*}
\text{sqr}[2n - r_1 - 1 : 0] &= \text{trunc}(t^2, e_1) \\
\text{cube}[3n - r_2 - 1 : 0] &= \text{trunc}(t \times (\text{sqr} << r_1), e_2) \\
\hat{p}[4n - r_3 - 1 : 0] &= \text{trunc}(((a \times \text{cube}) << r_2) + ((b \times \text{sqr}) << (n + r_1)) + ((c \times t) << 2n) + (d << 3n), e_3)
\end{align*}
\]

The result of exploiting the errors defined in equations (6.6) for each of these SOPs is shown in Figures 6.3, 6.4 and 6.5 in the case when \( n = 16 \) which uses Booth radix-4 and squarer arrays described in Section 6.1. The unfilled circles represent partial product bits which have been removed by the truncation, constant ones have been added within the partial product array and finally the least significant \( q_i \) columns are removed after each array has been summed.

Now in order to implement the heuristic, a value must be given to \( \frac{\partial R_i}{\partial e_i} \), namely the rate of change of the number of partial product bits removed with respect to the error. Small changes in the errors are unlikely to alter the output bit width of the operator, therefore it is assumed that error changes
only affect the number of partial product bits in the given operator. A value can be given to $\frac{\partial R_j}{\partial e_i}$ by considering the extra amount of error required to remove another partial product bit from the operator array. The method for truncating an arbitrary array described on page 187 states that with an error of the form $u2^r$ a set of bits $\Delta$ is removed from the array whose value is bounded by:

$$\text{val}(\Delta) < (2u - 1)2^r = 2e - 2^\lceil \log_2 e \rceil$$

If the truncation removes $k$ columns from the array then removing another partial product bit would requiring increasing $\text{val}(\Delta)$ by $2^k$ and for the bound to still be satisfied, $e$ must be increased by $2^{k-1}$. So a value for $\frac{\partial R_j}{\partial e_i}$ is:

$$\frac{\partial R_j}{\partial e_i} = \begin{cases} \frac{1}{2^{k_i}} & j = i \\ 0 & \text{otherwise} \end{cases}$$

where $k_i$ is the number of truncated columns in the $i$th operator. Simplifying
the algorithm held within equations (6.5) in light of this heuristic produces
the following procedure which will work whenever $\hat{p}$ is linear in the variables
$\varepsilon_i$ and the DFG only consists of SOPs:

- Set $e_i = \eta/(\lambda_i n)$ where $n$ is the number of operators.
- Compute the number of truncated columns $k_i$ which results in exploit-
ing error $e_i$ in operator $i$.
- While the following are not equal for all $i$:

\[
\left\{ \lambda_i 2^{k_i} \right\}
\]

Replace $e_j$ by the following:

\[
\frac{\eta e_j 2^{-k_j}}{\lambda_j \sum_{i} e_i 2^{-k_i}}
\]

**Bilinear Interpolation**

The second example comes from the 2D interpolation of four unsigned $n$ bit
integer values $a$, $b$, $c$ and $d$ which requires calculating:

\[
p = (1 - t)(1 - s)a + (1 - t)sb + t(1 - s)c + tsd
\]

The interpolating values $t$ and $s$ are in the interval $[0, 1]$, this interval can
be achieved by replacing $t$ and $s$ with $t/(2^n - 1)$ and $s/(2^n - 1)$ respectively,
where $t$ and $s$ are now $n$ bit integers. The equation then becomes:

\[
p = \left(1 - \frac{t}{2^n - 1}\right) \left(1 - \frac{s}{2^n - 1}\right) a + \left(1 - \frac{t}{2^n - 1}\right) \frac{s}{2^n - 1} b
\]

\[
+ \frac{t}{2^n - 1} \left(1 - \frac{s}{2^n - 1}\right) c + \frac{t}{2^n - 1} \frac{s}{2^n - 1} d
\]

\[
= \frac{7a + 7sb + 7sc + tsd}{(2^n - 1)^2}
\]

where $7$ is the bit wise inversion of $t$. A natural overall accuracy requirement
is that of allowing an absolute error of at most 1 which will return one of
the nearest integers, hence in this case $\eta = 1$. An example architecture that
is now used is interpolation with respect to \( s \) followed by \( t \):

\[
\begin{align*}
    s_{a,b} &= \frac{sa + sb}{2^n - 1} \\
    s_{c,d} &= \frac{sc + sd}{2^n - 1} \\
    p &= \frac{ts_{a,b} + ts_{c,d}}{2^n - 1}
\end{align*}
\]

This has an operator graph shown in Figure 6.6. This architecture has six

![Figure 6.6: DFG of Bilinear Interpolation.](image)

operators and thus has six potential locations for errors to be introduced. The maximum absolute allowable error is again denoted by \( \epsilon_i \) and is indicated in Figure 6.6. If the actual error at each node is \( \epsilon_i \) then the actual value of the implementation returns the following value:

\[
\begin{align*}
    \hat{s}_{a,b} &= \frac{sa + sb + \epsilon_1}{2^n - 1} + \epsilon_3 \\
    \hat{s}_{c,d} &= \frac{sc + sd + \epsilon_2}{2^n - 1} + \epsilon_4 \\
    \hat{p} &= \frac{ts_{a,b} + ts_{c,d} + \epsilon_5}{2^n - 1} + \epsilon_6
\end{align*}
\]

This design also has \( \hat{p} \) which is linear in variables \( \epsilon_i \) which means that the linear version of the iterative heuristic algorithm held within equations (6.5) is still appropriate. However, this design requires exploiting error freedom
in constant division. If the $e_i$ variables are written as follows:

\[
e_1 = u_1 \times 2^{r_1} \quad u_1 \in (1/2, 1]
\]
\[
e_2 = u_2 \times 2^{r_2} \quad u_2 \in (1/2, 1]
\]
\[
e_3 = u_3 \times 2^{r_3} \quad u_3 \in [1, 2)
\]
\[
e_4 = u_4 \times 2^{r_4} \quad u_4 \in [1, 2)
\]
\[
e_5 = u_5 \times 2^{r_5} \quad u_5 \in (1/2, 1]
\]
\[
e_6 = u_6 \times 2^{r_6} \quad u_6 \in [1, 2)
\]

This translates into a design with the following bit widths:

\[
SOP_1[2n - r_1 - 1 : 0] = \text{trunc}(\hat{s}a + sb, e_1)
\]
\[
SOP_2[2n - r_2 - 1 : 0] = \text{trunc}(\hat{s}c + sd, e_2)
\]
\[
\hat{s}_{a,b}[n - 1 - r_3 : 0] = \text{trunc}\left(\frac{2^{r_1 - r_3}SOP_1}{2^n - 1} \right)
\]
\[
\hat{s}_{c,d}[n - 1 - r_4 : 0] = \text{trunc}\left(\frac{2^{r_2 - r_4}SOP_2}{2^n - 1} \right)
\]
\[
SOP_3[2n - r_5 - 1 : 0] = \text{trunc}(\hat{t}s_{a,b} + t\hat{s}_{c,d}, e_5)
\]
\[
\hat{p}[n - 1 - r_6 : 0] = \text{trunc}\left(\frac{2^{r_5 - r_6}SOP_3}{2^n - 1} \right)
\]

The first step in applying the heuristic algorithm held within equations (6.5) is the calculation of the values of $\lambda_i$:

\[
\lambda_1 = \max \left| \frac{\partial \hat{p}}{\partial \epsilon_1} \right| = \max \left| \frac{\hat{t}}{(2^n - 1)^2} \right| = \frac{1}{2^n - 1}
\]
\[
\lambda_2 = \max \left| \frac{\partial \hat{p}}{\partial \epsilon_2} \right| = \max \left| \frac{t}{(2^n - 1)^2} \right| = \frac{1}{2^n - 1}
\]
\[
\lambda_3 = \max \left| \frac{\partial \hat{p}}{\partial \epsilon_3} \right| = \max \left| \frac{\hat{t}}{2^n - 1} \right| = 1
\]
\[
\lambda_4 = \max \left| \frac{\partial \hat{p}}{\partial \epsilon_4} \right| = \max \left| \frac{t}{2^n - 1} \right| = 1
\]
\[
\lambda_5 = \max \left| \frac{\partial \hat{p}}{\partial \epsilon_5} \right| = \max \left| \frac{1}{2^n - 1} \right| = \frac{1}{2^n - 1}
\]
\[
\lambda_6 = \max \left| \frac{\partial \hat{p}}{\partial \epsilon_6} \right| = 1
\]
The starting values of $e_i$ when applying the heuristic algorithm held within equations (6.5) are $e_i = \frac{n}{X^{kn}}$ where $n$ is the number of operators, which, in this case, is six. The starting values of $e_i$ are then the following:

$$
\begin{align*}
  e_1 &= e_2 = e_5 = \frac{2^n - 1}{6} \\
  e_3 &= e_4 = e_6 = \frac{1}{6}
\end{align*}
$$

This is the starting state, in order to implement the iterative algorithm values for $\frac{\partial R_j}{\partial e_i}$ must be calculated. A value has already been given to $\frac{\partial R_j}{\partial e_i}$ for a SOP, it is thus required to establish a value for constant division.

Recall from the previous chapter on page 179 that the constant division of unsigned $m$ bit number $x$ by $d$, implemented as $(ax + b) >> k$ is completely determined by the value $k$. For a faithfully rounded scheme, the optimal value of $k$ is defined as follows:

$$
\begin{align*}
  k_{opt} &= \min(k^+, k^-) \\
  k^\pm &= \min\left(k : \frac{2^k}{(\mp 2^k) \mod d} > \left\lfloor \frac{2^n}{d} \right\rfloor \right)
\end{align*}
$$

Allowable error introduced into the constant division is via dividing $x$ by $2^r$ for some $r < 0$. The value of $k$ used must then satisfy:

$$
\begin{align*}
  k_{opt}^\pm &= \min\left(k : \frac{2^k}{(\mp 2^k) \mod d} > \left\lfloor \frac{2^{m-r}}{d} \right\rfloor \right)
\end{align*}
$$

If the error is doubled then $r$ increases by one and $k_{opt}$ will change if one of the following conditions holds:

$$
\frac{2^{k_{opt} - 1}}{(\mp 2^{k_{opt} - 1}) \mod d} > \left\lfloor \frac{2^{m-r-1}}{d} \right\rfloor
$$

If this condition holds, $k$ would be decreased, which will remove one partial product row from the product $ax$, removing $m$ bits. Whether or not this condition is satisfied, the output of the constant divisor has been reduced by 1 bit. If the result of the constant division is subsequently used in another operator, further partial product bits will be removed. Let $n_{i,j}$ denote the bit width of the variable that the output of operator $i$ is multiplied by in operator $j$. So if operator $i$ is a constant division operator with input bit
width \( n_{q,i} \) for some \( q \), the constant for division is \( d_i \), \( k_{opt} \) for the operator is \( k_i \), the permissible maximum absolute error if \( e_i \) and \( r_i = \lfloor \log_2(e_i) \rfloor \) then a value for \( \frac{\partial R_j}{\partial e_i} \) can be given as follows:

\[
\frac{\partial R_j}{\partial e_i} = \begin{cases} \frac{n_{q,i}}{2e_i} & j = i \text{ and } \text{cond holds} \\ 0 & j = i \text{ and } \text{cond doesn’t hold} \\ \frac{n_{q,i}}{2e_i} & \text{otherwise} \end{cases}
\]

\[
\text{cond} = \left( \frac{2^{k_i-1}}{\pm 2^{k_i-1}} \right) \mod d_i > \left\lfloor \frac{2n_{q,i} - r_i - 1}{d_i} \right\rfloor
\]

Simplifying the algorithm held within equations (6.5) in light of this new heuristic produces the following procedure which will work whenever \( \hat{p} \) is linear in the variables \( \epsilon_i \) and the DFG consists of SOPs and constant divisions:

- Set \( \epsilon_i = \eta/(\lambda_i n) \) where \( n \) is the number of operators.
- Compute the number of truncated columns \( k_i \) which results in exploiting error \( e_i \) in SOP \( i \).
- Compute the constant shift value \( k_i \) used in exploiting error \( e_i \) in constant division \( i \).
- Compute \( \frac{\partial R_j}{\partial e_i} \), when operator \( j \) is a SOP this calculation is:

\[
\frac{\partial R_j}{\partial e_i} = \begin{cases} \frac{1}{2^{k_i-1}} & j = i \\ 0 & \text{otherwise} \end{cases}
\]

When operator \( j \) is a constant division this calculation is:

\[
\frac{\partial R_j}{\partial e_i} = \begin{cases} \frac{n_{q,i}}{2e_i} & j = i \text{ and } \text{cond holds} \\ 0 & j = i \text{ and } \text{cond doesn’t hold} \\ \frac{n_{q,i}}{2e_i} & \text{otherwise} \end{cases}
\]

\[
\text{cond} = \left( \frac{2^{k_i-1}}{\pm 2^{k_i-1}} \right) \mod d_i > \left\lfloor \frac{2n_{q,i} - r_i - 1}{d_i} \right\rfloor
\]

- While the following are not equal for all \( j \):

\[
\left\{ \frac{\lambda_j}{\sum_i \frac{\partial R_i}{\partial e_j}} \right\}
\]

205
Replace $e$ by the following:

$$\frac{\eta \sum_i e_j \frac{\partial R_i}{\partial e_j}}{\lambda \sum_{i,j} e_j \frac{\partial K}{\partial e_j}}$$

### Gabor Filter

The last example investigates the situation where $\hat{p}$ is non linear in variables $\varepsilon$. It concerns the implementation of an instance of a Gabor filter [52]:

$$e^{-(a^2+b^2)} \approx p = 1 - (a^2 + b^2) + \frac{(a^2 + b^2)^2}{2} - \frac{(a^2 + b^2)^3}{6} + \frac{(a^2 + b^2)^4}{24}$$

If $a$ and $b$ are $n$ bit fixed point unsigned numbers in the interval $[0, 1)$ then this polynomial can be assumed to have integer inputs if $a$ and $b$ are replaced by $a/2^n$ and $b/2^n$. This results in the following polynomial:

$$p = \frac{1}{2^{8n+3}} \left( 2^{8n+3} - 2^{6n+3}(a^2 + b^2) + 2^{4n+2}(a^2 + b^2)^2 \right. \\
- 2^{2n+2} \left( \frac{a^2 + b^2)^3}{3} + \frac{(a^2 + b^2)^4}{3} \right)$$

A natural overall accuracy requirement is that of allowing an overall error equal to the precision of the inputs, namely $2^{-n}$. This can be achieved by seeking an accuracy of $\eta = 2^{7n+3}$ of the following polynomial:

$$p = 2^{8n+3} - 2^{6n+3}(a^2 + b^2) + 2^{4n+2}(a^2 + b^2)^2 \\
- 2^{2n+2} \left( \frac{a^2 + b^2)^3}{3} + \frac{(a^2 + b^2)^4}{3} \right)$$

Rewriting this polynomial as follows gives rise to an example architecture:

$$t = a^2 + b^2$$
$$t_2 = t^2$$
$$p = 2^{8n+3} - 2^{6n+3}t + 2^{4n+2}t_2 - 2^{2n+2} \frac{t \times t_2}{3} + \frac{t_2^2}{3}$$

206
Finally, writing this as a combination of SOPs and constant divisions produces the following architecture:

\[
\begin{align*}
t & = a^2 + b^2 \\
t_2 & = t^2 \\
t_3 & = 2^{2n+2} t \times t_2 - t_2^2 \\
t_4 & = \frac{t_3}{3} \\
p & = 2^{8n+3} - 2^{6n+3} t + 2^{4n+2} t_2 - t_4
\end{align*}
\]

This has an operator graph of a form found in Figure 6.7. This architecture has five operators and thus has five potential locations for errors to be introduced. The maximum absolute allowable error is again denoted by \( e_i \) and is indicated in Figure 6.7. If the actual error at each node is \( \varepsilon_i \) then the actual value of the implementation returns the following value:

\[
\begin{align*}
\hat{t} & = a^2 + b^2 + \varepsilon_1 \\
\hat{t}_2 & = t^2 + \varepsilon_2 \\
\hat{t}_3 & = 2^{2n+2} \hat{t} \times \hat{t}_2 - \hat{t}_2^2 + \varepsilon_3 \\
\hat{t}_4 & = \frac{\hat{t}_3}{3} + \varepsilon_4 \\
\hat{p} & = 2^{8n+3} - 2^{6n+3} t + 2^{4n+2} t_2 - t_4 + \varepsilon_5
\end{align*}
\]
Calculating the difference between \( p \) and \( \hat{p} \) returns the following polynomial:

\[
p - \hat{p} = \sum_{\alpha} c_{\alpha}(x)e^\alpha
\]

\[
= -\frac{\varepsilon_4^3}{3} - \frac{4(t - 2^{2n})\varepsilon_1^3}{3} - \frac{2\varepsilon_1^2\varepsilon_2}{3} - \frac{2(t^2 - 2^{2n+1}t + 2^{4n+1})\varepsilon_1^2}{3}
\]

\[
- \frac{4(t - 2^{2n})\varepsilon_1\varepsilon_2}{3} - \frac{4(t^3 - 3 \times 2^{2n}t^2 + 3 \times 2^{4n+1}t - 3 \times 2^{6n+1})\varepsilon_1}{3} - \frac{\varepsilon_2^2}{3}
\]

\[
- \frac{2(t^2 - 2^{2n+1}t + 3 \times 2^{4n+1})\varepsilon_2}{3} + \varepsilon_3 + \varepsilon_4 - \varepsilon_5
\]

Applying the heuristic requires computing the polynomial \( \sum_{\alpha} \lambda_{\alpha}e^\alpha \) where the \( \lambda \) coefficients are the maximum absolute value of the coefficients of \( p - \hat{p} \) over all possible values of \( t = a^2 + b^2 \in [0, 2(2^n - 1)^2] \). Doing so results in the following polynomial:

\[
\sum_{\alpha} \lambda_{\alpha}e^\alpha = \frac{e_4^4}{3} + \frac{2^{2n+2}e_1^3}{3} + \frac{2e_1^2e_2}{3} + 2^{4n+2}e_1^2 + \frac{2^{2n+2}e_1e_2}{3}
\]

\[
+ 2^{6n+3}e_1 + \frac{e_2^2}{3} + 2^{4n+2}e_2 + \frac{e_3}{3} + e_4 + e_5
\]

The first stage in applying the heuristic iterative algorithm held within equations (6.3) is to solve the following set of equations:

\[
\sum_{\alpha} \lambda_{\alpha}e_j \frac{\partial e^\alpha}{\partial e_j} = T \quad \text{for all } j
\]

\[
\sum_{\alpha} \lambda_{\alpha}e^\alpha = \eta
\]

The first set of these equations is as follows:

\[
\frac{4e_1^4}{3} + 2^{2n+2}e_1^3 + \frac{4e_1^2e_2}{3} + 2^{4n+3}e_1^2 + \frac{2^{2n+2}e_1e_2}{3} + 2^{6n+3}e_1 = T
\]

\[
\frac{2e_1^2e_2}{3} + \frac{2^{2n+2}e_1e_2}{3} + 2e_2^2 + 2^{4n+2}e_2 = T
\]

\[
\frac{e_3}{3} = T
\]

\[
e_4 = T
\]

\[
e_5 = T
\]
These equations along with $\sum_{\alpha} \lambda_{\alpha} e^{\alpha} = \eta$ can be solved via Gröbner bases techniques. In the case of $n = 16$ the solution which satisfies $e_i \geq 0$ is:

\[
e_1 = 13107.154666879 \\
e_2 = 0.11258983 \times 10^{15} \\
e_3 = 0.249230401 \times 10^{35} \\
e_4 = 0.830768004 \times 10^{34} \\
e_5 = 0.830768004 \times 10^{34}
\]

If the errors are of the following form:

\[
e_1 = u_1 \times 2^{r_1} \quad u_1 \in (1/2, 1] \\
e_2 = u_2 \times 2^{r_2} \quad u_2 \in (1/2, 1] \\
e_3 = u_3 \times 2^{r_3} \quad u_3 \in (1/2, 1] \\
e_4 = u_4 \times 2^{r_4} \quad u_4 \in [1, 2) \\
e_5 = u_5 \times 2^{r_5} \quad u_5 \in (1/2, 1]
\]

Then the implementation will have the following bit widths:

\[
\hat{t}[2n - r_1 : 0] = \text{trunc}(a^2 + b^2, e_1) \\
\hat{t}_2[4n - r_2 + 1 : 0] = \text{trunc}(2^{r_1 \hat{t}_2^2}, e_2) \\
\hat{t}_3[8n - r_3 + 3 : 0] = \text{trunc}(2^{2n+r_1+r_2+2\hat{t}_2} \times \hat{t}_2 - 2^{2r_2 \hat{t}_2^2}, e_3) \\
\hat{t}_4[8n - r_4 + 2 : 0] = \text{trunc}(\frac{\hat{t}_32^{r_3}}{3}, e_4) \\
\hat{p}[8n - r_5 + 3 : 0] = \text{trunc}(2^{8n+3} - 2^{6n+r_1+3\hat{t}_2} + 2^{4n+r_2+2\hat{t}_2} - 2^{r_4 \hat{t}_4}, e_5)
\]

In order to implement the iterative step of the algorithm held within equations (6.3), the current errors $e_i$ need to be updated to variables $f_i$ where:

\[
f_j \sum_{\alpha} \lambda_{\alpha} \frac{\partial f^{\alpha}}{\partial f_j} = T e_j \sum_{i} \frac{\partial R_i}{\partial e_j} \quad \text{for all } j \\
\sum_{\alpha} \lambda_{\alpha} f^{\alpha} = \eta
\]
Which in this case becomes the following equations:

\[
\begin{align*}
\frac{4f_1^4}{3} + 2^{2n+2}f_1^3 + \frac{4f_1^2 f_2}{3} + 2^{4n+3}f_2^2 + \frac{2^{2n+2} f_1 f_2}{3} + 2^{6n+3}f_1 &= Te_1 \sum_i \frac{\partial R_i}{\partial e_1} \\
\frac{2f_1^2 f_2}{3} + \frac{2^{2n+2} f_1 f_2}{3} + \frac{2f_2^2}{3} + 2^{4n+2}f_2 &= Te_2 \sum_i \frac{\partial R_i}{\partial e_2} \\
\frac{f_3}{3} &= Te_3 \sum_i \frac{\partial R_i}{\partial e_3} \\
f_4 &= Te_4 \sum_i \frac{\partial R_i}{\partial e_4} \\
f_5 &= Te_5 \sum_i \frac{\partial R_i}{\partial e_5} \\
\frac{f_1^4}{3} + \frac{2^{2n+2} f_1^3}{3} + \frac{2f_1^2 f_2}{3} + 2^{4n+2}f_2^2 + \frac{2^{2n+2} f_1 f_2}{3} \\
&+ 2^{6n+3}f_1 + \frac{f_2^2}{3} + 2^{4n+2}f_2 + \frac{f_3^3}{3} + f_4 + f_5 = 2^{7n+3}
\end{align*}
\]

Given the current values of \(e_i\) and \(\frac{\partial R_i}{\partial e_j}\), this set of polynomial equations can be solved via Gröbner bases techniques to establish the updated error values \(f_i\). This now establishes how the iterative algorithm deals with the case when \(\hat{p}\) is non linear in variables \(\varepsilon\) and given that the previous two examples gave value to \(\frac{\partial R_i}{\partial e_j}\) in the case of a SOP and constant division operator it is now possible to give a full description of the iterative heuristic algorithm initially put forward in the equations (6.3). This is given in the next section.

**6.3.3. Procedure for Error Landscape Navigation**

For the general case it is assumed that \(p\) has rational coefficients and integer inputs. If an input is actually fixed-point then, as in the case of the cubic polynomial and Gabor filter examples, it can be replaced by \(x/2^n\) for some integer \(n\), where \(x\) is now an integer input. It is also assumed that a DFG exists consisting of sum-of-product operations or constant division operators. SOP operations are functions of the form:

\[
\sum_i (-1)^s_i 2^{r_i} x_i y_i
\]
where \( s_i \) is a constant zero or one, \( r_i \) are constant non-negative integers and \( x_i \) and \( y_i \) are integer variable inputs or fixed-point intermediate variables. Such SOPs can be embodied in a single array with \( r_i \) aligning the products with respect to each other. It is also assumed that bounds on each intermediate variable are known. As discussed on page 49, iterative use of Satisfiability Modulo Theories (SMT) can provide tight bounds but with the risk of the computational complexity resulting in the SMT solvers being unable to provide proofs in reasonable time. However their use is iterative, with a bound being produced at each stage which becomes refined, thus early termination still provides a bound [98]. These bounds allow for bit widths which are just large enough to hold the values they represent and mean that prior to error allocation and exploitation, operator cost is reduced. Under these assumptions the procedure for error landscape navigation can now be presented.

- Given a polynomial \( p \) and an architecture consisting of operations which are SOPs or constant divisions, construct \( \hat{p} \) which has additive error \( \varepsilon_i \) for each operator.

- Expand \( p - \hat{p} \) with respect to variables \( \varepsilon_i \), producing an expansion of the form:

\[
p - \hat{p} = \sum_{\alpha} c_{\alpha}(x) \varepsilon^{\alpha}
\]

- Calculate the variables \( \lambda_{\alpha} \) which are defined as follows (this can be achieved by iterative use of SMT as described in [98]):

\[
\lambda_{\alpha} = \max_x |c_{\alpha}(x)|
\]

- Either use a given starting maximum absolute error bound for each operator \( \varepsilon_i \) or use a default one by solving the following equations via Gröbner bases which includes the maximum absolute error tolerance
defined by the user $\eta$:

$$\sum_{\alpha} \lambda_{\alpha} e_j \frac{\partial e_{\alpha}}{\partial e_j} = T \quad \text{for all } j$$

$$\sum_{\alpha} \lambda_{\alpha} e_{\alpha} = \eta \quad \text{(6.7)}$$

- For each SOP operator with associated maximum absolute allowable error $e_i$ write $e_i$ as $e_i = u_i 2^{r_i}$ where $u_i \in (1/2, 1]$. Exploiting the error freedom will result in reducing the output bit width by $r_i$.

- For each constant division operator with associated maximum absolute allowable error $e_i$ write $e_i$ as $e_i = u_i 2^{r_i}$ where $u_i \in [1, 2)$. Exploiting the error freedom will result in reducing the output bit width by $r_i$.

- Calculate the new bit widths of the resultant DFG $n_{i,j}$ which is the bit width of the intermediate variables emerging from operator $i$ into operator $j$.

- For each SOP operator create the partial product array and using the technique put forward in equation (6.1.1) calculate the number of truncated columns $k_i$.

- For each constant division operator which requires the division by $d_i$ implement a faithfully rounding of $x/(2^{r_i} d_i)$ via the method put forward in equations (6.1) and calculate the associated shift value $k_i$.

- Compute $\frac{\partial R_j}{\partial e_i}$ for each SOP operator $j$:

$$\frac{\partial R_j}{\partial e_i} = \begin{cases} 
\frac{1}{2^{n_i-1}} & j = i \\
0 & \text{otherwise} 
\end{cases}$$

- Compute $\frac{\partial R_j}{\partial e_i}$ for each constant division operator $j$ where $n_{q,i}$ is the
input bit width into the operator:

\[
\frac{\partial R_j}{\partial e_i} = \begin{cases} 
\frac{n_{q,i}}{2e_i} & j = i \text{ and } \text{cond} \text{ holds} \\
0 & j = i \text{ and } \text{cond} \text{ doesn’t hold} \\
\frac{n_{q,i}}{2e_i} & \text{otherwise}
\end{cases}
\]

\[
\text{cond} = \left\lfloor \frac{2^{k_i-1} \left( \mp 2^{k_i-1} \right) \mod d_i}{2^{n_{q,i}-r_i-1}} \right\rfloor
\]

- While the following are not equal for all \( j \) (alternatively the algorithm can be iterated for a maximum number of iterations and the iteration that has the smallest range of these values can be returned):

\[
\left\{ \frac{1}{\sum_i \frac{\partial R_i}{\partial e_j}} \sum_\alpha \lambda_\alpha \frac{\partial e^\alpha}{\partial e_j} \right\}
\]  

(6.8)

Replace \( e \) by \( f \) by solving the following equations for \( f \) using Gröbner bases:

\[
f_j \sum_\alpha \lambda_\alpha \frac{\partial f^\alpha}{\partial f_j} = T e_j \sum_i \frac{\partial R_i}{\partial e_j} \quad \text{for all } j
\]

\[
\sum_\alpha \lambda_\alpha f^\alpha = \eta
\]  

(6.9)

This procedure requires array creation for the SOP operators, the examples have used Booth radix-4 and squarer arrays which were discussed in Section 6.1. However, the procedure can be used where the operators can be implemented via the summation of any binary array. In particular there are a variety of differing multiplier architectures which result in the summation of a binary array, with which this procedure can be used.
6.4. Fixed-Point Polynomial Hardware Heuristic Cost Function

Given the plethora of potential different DFGs that can be created for the same polynomial, combined with the multitudinous ways in which the errors can be assigned to each operator, there are numerous potential implementation candidates. Applying logic synthesis to all of them is time consuming and costly, moreover the iterative algorithm could be augmented by early feedback on the quality of the designs that are being produced. This early feedback can be provided by a hardware heuristic cost function which should be simple to evaluate and sound in its discrimination. In this section, such a heuristic is developed. A heuristic for the area of the implementation has shown to be reliable, as such, an area heuristic is presented here. This heuristic cost function is developed using a unit area cost model for AND and OR gates and a unit area cost of two for XOR gates and is developed here by starting with a binary integer adder.

6.4.1. Binary Integer Adder

The starting point for the heuristic cost function is an integer adder with a logarithmic construction. A reference point is taken from the background material in Chapter 2 on page 36 and use a parallel prefix architecture [149]. Consider the sum, $s$, of two $w$ bit inputs $a$ and $b$. Letting $a_i$, $b_i$ and $s_i$ denote the $i$th bit of $a$, $b$ and $s$ respectively then $s_i$ can be expressed as:

$$s_i = x_i \oplus (g_{i-1} \lor (p_{i-1} \land g_{i-2}) \lor (p_{i-1} \land p_{i-2} \land g_{i-3}) \lor \ldots \lor (p_{i-1} \land p_{i-2} \ldots p_1 \land g_0))$$

where $g_i = a_i \land b_i$ $p_i = a_i \lor b_i$ $x_i = a_i \oplus b_i$

where $\lor$, $\land$ and $\oplus$ denote logic operations OR, AND and XOR respectively.

A parallel prefix architecture uses the following intermediate variables:

$$P_{i-1,0} = p_{i-1} \land p_{i-2} \land \ldots \land p_1$$

$$G_{i-1,0} = g_{i-1} \lor (p_{i-1} \land g_{i-2}) \lor (p_{i-1} \land p_{i-2} \land g_{i-3}) \lor \ldots \lor (p_{i-1} \land p_{i-2} \ldots p_1 \land g_0)$$

214
Recursive formulae for these variables are as follows:

\[ P_{i-1,0} = P_{i-1,j} \land P_{j-1,0} \]
\[ G_{i-1,0} = G_{i-1,j} \lor (P_{i-1,j} \land G_{j-1,0}) \]

These recursive definitions of \( P \) and \( G \) are used to logarithmically create \( G_{i-1,0} \) for all \( i \):

\[ s_i = x_i \oplus G_{i-1,0} \]

The delay and area cost of the logarithmic steps involved in the adder are (using cost one for AND and OR gates and cost two for XOR gates) is summarised in Table 6.1.

### 6.4.2. Binary Rectangular Arrays

In the case of a parallel reduction of a rectangular array of bits, reduction cells, such as a Full Adder (FA), are used to reduce the array to a point at which an integer adder can be used. Full adders sum three bits, the equations for the two bit result can be written as:

\[ \text{sum} = (a \oplus b) \oplus c \]
\[ \text{carry} = a \land b \lor (c \land (a \oplus b)) \]

Using the unit cost model, a full adder has delay 4 and area 7. Figure 6.8 shows how repeated use of full adders can be used to reduce a rectangular array of height 6. At each level the height of the array is reduced by 2/3, so in general the area cost of summing \( h \), \( w \) bit numbers (denoting this as

---

**Table 6.1.: Area cost of Parallel Prefix Adder.**

<table>
<thead>
<tr>
<th>Signal Creation</th>
<th>Area Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>( g, p ) and ( x )</td>
<td>4w</td>
</tr>
<tr>
<td>( G_{2i+1,2i} ) ( P_{2i+1,2i} ) for all ( i )</td>
<td>3w/2</td>
</tr>
<tr>
<td>( G_{4i+3,2i} ) ( G_{4i+2,2i} ) ( P_{4i+3,2i} ) ( P_{4i+2,2i} ) for all ( i )</td>
<td>3w/2</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( G_{w-1,0} ) ( G_{w-2,0} ) ( \ldots ) ( G_{w/2,0} )</td>
<td>( w )</td>
</tr>
<tr>
<td>( s_i ) for all ( i )</td>
<td>2w</td>
</tr>
<tr>
<td>Total</td>
<td>( \frac{w}{2} (3</td>
</tr>
</tbody>
</table>
Figure 6.8.: Reduction of a Binary Rectangular Array.

\[ \text{area}(\text{sum}(h, w)) \] can be computed recursively as:

\[ \text{area}(\text{sum}(h, w)) \approx \frac{wh}{3} \times \text{FA} + \text{area}\left(\text{sum}\left(\frac{2h}{3}, w\right)\right) \]
\[ \approx \frac{wh}{3} \times \text{FA} + \frac{2wh}{3} \times \text{FA} + \ldots + \left(\frac{2}{3}\right)^{f-1} \times \frac{wh}{3} \times \text{FA} + \text{area}(\text{sum}(2, w)) \]
\[ \approx \left(1 - \left(\frac{2}{3}\right)^f\right) \times wh \times \text{FA} + \text{area}(\text{sum}(2, w)) \]

where

\[ 2 \approx \left(\frac{2}{3}\right)^f \quad f = \left\lceil \frac{\log_2 h - 1}{\log_2 3 - 1} \right\rceil \]

Assuming full adders have area 7 and using Section 6.4.1:

\[ \text{area}(\text{sum}(h, w)) \approx 7wh \left(1 - \left(\frac{2}{3}\right)^f\right) + \frac{w}{2} \left(3\left\lceil \log_2 w \right\rceil + 11\right) \]

where \( f = \left\lceil \frac{\log_2 h - 1}{\log_2 3 - 1} \right\rceil \)

Note that these formulae simplify to the binary integer adder case when \( h = 2 \).
6.4.3. Arbitrary Binary Array

In the case of an arbitrary binary array, its important characteristics are its width $w$, height $h$ and the number of bits within the array $pp$. The argument from the previous section can be lifted, using the fact that $wh$ can now be replaced by $pp$ giving rise to:

$$\text{area} (\text{array} (h, w, pp)) \approx 7pp \left( 1 - \left( \frac{2}{3} \right)^f \right) + \frac{w}{2} (3 \lfloor \log_2 w \rfloor + 11)$$

where $f = \left\lceil \frac{\log_2 h - 1}{\log_2 3 - 1} \right\rceil$ (6.10)

The cost of implementing an arbitrary binary array is a crucial object upon which other heuristics is built.

6.4.4. Multipliers

The heuristic cost of array reduction has been established, this leaves the question of array creation cost. Recall from the background material on page 40 that the formulation of Booth radix-4 multiplication derives from the following observation:

$$ab = a \left( \sum_{i=0}^{i=m-1} b_i 2^i \right) = \sum_{i=0}^{i=m/2} \left( a (b_{2i-1} + b_{2i} - 2b_{2i+1}) \right) 2^{2i}$$

Noting that $b_{2i-1} + b_{2i} - 2b_{2i+1} \in \{-2, -1, 0, 1, 2\}$ and these multiples of $a$ are easy to produce (in redundant form in the case of $-a = \bar{a} + 1$). Figure 6.9 shows the multiplication array for an unsigned 10 bit Booth radix-4 multiplication. A bit in the array can be formulated as:

$$pp_{i,j} = ((a_j \oplus b_{2i+1}) \land (b_{2i} \oplus b_{2i-1})) \lor (a_{j-1} \oplus b_{2i+1}) \land (b_{2i} \oplus b_{2i-1}) \land (b_{2i+1} \oplus b_{2i})$$
An efficient implementation of the array requires the creation, for $i = 0 \ldots \lfloor m/2 \rfloor$, of:

\[
\begin{align*}
t_{0i} &= a \oplus b_{2i+1} \\
t_{1i} &= b_{2i} \oplus b_{2i-1} \\
t_{2i} &= b_{2i+1} \oplus b_{2i} \\
t_{3i} &= \overline{t_{1i}} \land t_{2i}
\end{align*}
\]

Followed by for $i = 0 \ldots \lfloor m/2 \rfloor$ and $j = 0 \ldots n$:

\[
pp_{i,j} = (t_{0ij} \land t_{1i}) \lor (t_{0i-1j} \land t_{3i})
\]

Hence using the cost model provides the following Booth multiplier heuristic cost:

\[
area(mult_{Booth}(n, m)) \approx 5nm/2 + area(array(m/2, n + m, nm/2))
\]

### 6.4.5. Squarers

Using the squarer construction from Section 6.1 reduces a standard AND array of the following form in the case of a four bit input $a$:

\[
\begin{array}{cccccccc}
a_3 \land a_0 & a_2 \land a_0 & a_1 \land a_0 & a_0 \land a_0 \\
a_3 \land a_1 & a_2 \land a_1 & a_1 \land a_1 & a_0 \land a_1 \\
a_3 \land a_2 & a_2 \land a_2 & a_1 \land a_2 & a_0 \land a_2 \\
a_3 \land a_3 & a_2 \land a_3 & a_1 \land a_3 & a_0 \land a_3
\end{array}
\]

For an $n$ bit squarer, $n(n - 1)/2$ bits are duplicated and the simplified array consists of $n(n + 1)/2$ bits. Conclude that:

\[
area(squarer(n)) \approx n^2/2 + area(array(n/2, 2n, n^2/2))
\]
6.4.6. Constant Multiplication

Constant multiplication may occur within a SOP and does occur in the multiply add scheme of the constant division operators. Recall from Section 6.1 that a canonical signed digit implementation of a multiplication of variable \( x[m-1:0] \) by constant \( a[n-1:0] \) requires the creation of a signal \( c \), via the following algorithm:

\[
\text{Input } a[n-1:0] \quad \text{binary string to be encoded}
\]
\[
\text{Output } c[n:0] \quad \text{equivalent canonical signed digit representation}
\]

begin
\[
a_{-1} = 0
t_{-1} = 0
\]
for \( i = 0 \) to \( n \) {
\[
t_i = t_{i-1} \lor (a_i \oplus a_{i-1})
c_i = t_i? (a_{i+1}? 1 : 1) : 0
\]
end

The array width will be \( m + n \) and the height will be number of non zero entries in \( c \), this is the same as the Hamming weight of the intermediate variable \( t \), denoted \( Hamm(t) \). The number of partial product bits is approximately \( mHamm(t) \). In conclusion the heuristic cost function of multiplying \( x[m-1:0] \) by constant \( a[n-1:0] \) is as follows:

\[
area(ax) \approx area(array(Hamm(t), n + m, mHamm(t)))
\]

where \( t \) satisfies \( t_i = t_{i-1} \lor (a_i \oplus a_{i-1}) \).

6.4.7. Example Heuristic Calculation Cost

As an example, consider the heuristic area cost of implementing the set of SOPs associated with the cubic polynomial example shown within Figures 6.3, 6.4 and 6.5. Their width, height and partial product count is shown in Table 6.2:

Now the heuristic cost function for Booth multiplication and squarer ar-
Table 6.2.: Characteristics of the Partial Product Arrays of a Cubic Polynomial Implementation.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Height</th>
<th>Width</th>
<th>Partial Product Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>squarer</td>
<td>9</td>
<td>22</td>
<td>110</td>
</tr>
<tr>
<td>multiplier</td>
<td>10</td>
<td>24</td>
<td>144</td>
</tr>
<tr>
<td>SOP</td>
<td>31</td>
<td>25</td>
<td>465</td>
</tr>
</tbody>
</table>

rays return the following values:

\[ \text{area}(\text{mult}_{\text{Booth}}(n, m)) \approx \frac{5nm}{2} + \text{area}(\text{array}(m/2, n + m, nm/2)) \]
\[ \text{area}(\text{squarer}(n)) \approx \frac{n^2}{2} + \text{area}(\text{array}(n/2, 2n, n^2/2)) \]

So if a SOP is constructed from Booth multipliers and is subsequently truncated, resulting in an array with height, width and partial product bit count \( h, w \) and \( pp \) then the heuristic area cost would be:

\[ \text{area}(\text{array}_{\text{Booth}}(h, w, pp)) \approx 5pp + \text{area}(\text{array}(h, w, pp)) \]

Similarly if a SOP is constructed from squarers and is subsequently truncated, resulting in an array with height, width and partial product bit count \( h, w \) and \( pp \) then the heuristic area cost would be:

\[ \text{area}(\text{array}_{\text{squarer}}(h, w, pp)) \approx pp + \text{area}(\text{array}(h, w, pp)) \]

Recalling the heuristic cost of an array from equation (6.10):

\[ \text{area}(\text{array}(h, w, pp)) \approx 7pp \left(1 - \left(\frac{2}{3}\right)^f\right) + \frac{w}{2} (3\lceil \log_2 w \rceil + 11) \]

where \( f = \left\lceil \frac{\log_2 h - 1}{\log_2 3 - 1} \right\rceil \)

Then the total area cost of the cubic polynomial can be calculated, the intermediate steps can be found in Table 6.3.

As an example of calculating the heuristic cost of implementing a constant division, consider a faithful rounding of \( x/3 \) where \( x \) is 32 bits in length say, as may occur in an implementation of the Gabor filter example. The
Table 6.3.: Cubic Polynomial Heuristic Area Cost.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Height</th>
<th>Width</th>
<th>Partial Product Count</th>
<th>Array Creation Count</th>
<th>FA Count</th>
<th>Array Reduction Area</th>
<th>Total Area Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>squarer</td>
<td>9</td>
<td>22</td>
<td>110</td>
<td>110</td>
<td>4</td>
<td>904</td>
<td>1014</td>
</tr>
<tr>
<td>multiplier</td>
<td>10</td>
<td>24</td>
<td>144</td>
<td>720</td>
<td>4</td>
<td>1121</td>
<td>1841</td>
</tr>
<tr>
<td>SOP</td>
<td>31</td>
<td>25</td>
<td>465</td>
<td>2325</td>
<td>7</td>
<td>3389</td>
<td>5714</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8569</td>
</tr>
</tbody>
</table>

equations for the multiply add scheme in the case of faithful rounding were represented in equations (6.1) for dividing an $m$ bit number by $d$:

\[
(a, b, k) = (k^+ < k^-)? \quad (a^+, b^+, k^+) \quad : \quad (a^-, b^-, k^-) \\

k^\pm = \min \left( k : \frac{2^k}{(\mp 2^k) \mod d} > \left\lfloor \frac{2^m}{d} \right\rfloor \right) \\
a^+ = \left\lceil \frac{2^{k^+}}{d} \right\rceil \\
\quad a^- = \left\lfloor \frac{2^{k^-}}{d} \right\rfloor \\
b^+ = 0 \quad b^- = \minHamm \left( (2^{k^-} - a^- d)|2^m/d|, 2^{k^-} - 1 \right)
\]

Substituting $m = 32$ and $d = 3$:

\[
(a, b, k) = (k^+ < k^-)? \quad (a^+, b^+, k^+) \quad : \quad (a^-, b^-, k^-) \\
k^\pm = \min \left( k : \frac{2^k}{(\mp 2^k) \mod 3} > \left\lfloor \frac{2^{32}}{3} \right\rfloor \right) \\
a^+ = \left\lceil \frac{2^{k^+}}{3} \right\rceil \\
\quad a^- = \left\lfloor \frac{2^{k^-}}{3} \right\rfloor \\
b^+ = 0 \quad b^- = \minHamm \left( (2^{k^-} - 3a^-)/2^{32}/3, 2^{k^-} - 1 \right)
\]

Evaluating these expressions gives rise to the following:

\[
(a, b, k) = (k^+ < k^-)? \quad (a^+, b^+, k^+) \quad : \quad (a^-, b^-, k^-) \\
k^+ = 31 \quad k^- = 32 \\
a^+ = 715827883 \quad a^- = 1431655765 \\
b^+ = 0 \quad b^- = \minHamm (1431655765, 4294967295)
\]

So the optimal design is $(a, b, k) = (715827883, 0, 31)$. Now the heuristic
area cost for the constant multiplication of 32 bit $x$ by the 30 bit constant $715827883$ is:

$$area(715827883x) \approx area(array(H\text{amm}(t), 62, 32H\text{amm}(t)))$$

where $t$ satisfies $t_i = \overline{t_{i-1}} \lor (a_i \oplus a_{i-1})$. The binary form of $a$ and hence of $t$ is:

$$a = 1010101010101010101010101011$$
$$t = 1010101010101010101010101010101$$

The Hamming weight of $t$ is 16 so the final heuristic area cost is:

$$area(715827883x) \approx area(array(16, 62, 512)) \approx 4168$$

In this manner a heuristic area cost for an implementation of an architecture consisting of truncated SOPs and constant divisions using the multiply add scheme can be constructed. The accuracy of this heuristic cost function put forward in this section is established by the logic synthesis of multiple candidate designs within the following experiments section and ascertaining whether application of the heuristic cost function would have correctly predicted the results of the synthesis.
6.5. Experiments

The procedure in Section 6.3.3 was implemented using Singular [43] for the polynomial manipulations and Hector [165] for the SMT operations. The output of the procedure is HDL which includes constant division and truncated SOPs. The constant division HDL is a single piece of parameterisable code which performs the faithfully rounded division of an \( n \) bit input by constant \( d \). For the truncated SOPs a single piece of highly parameterisable HDL was created that constructed an array that implemented the following general SOP:

\[
\sum_i (-1)^{s_i} 2^{2^{-r_i}} x_i y_i
\]

This HDL handled the cases of multiplication, squarers and constant multiplication. The arrays created in this HDL used a Booth radix-4 architecture for the multipliers and the arrays described in Section 6.2 for the squarers and constant multiplication. To facilitate the truncation of this HDL, and any HDL containing an array, in the presence of a maximum absolute error bound utilising the bounded error array Theorem (6.1.1), a tool was created which performed the operations described in Figure 6.10. Here, Hector is used to prove whether each bit of the partial product array is possible to evaluate to one for any possible input. This enables the creation of a robust flow at the expense of the runtimes of potentially hundreds of small lemmas. However, all these lemmas for all the SOPs present in the DFG can be parallelised.

The three examples, cubic polynomial, bilinear interpolation and Gabor filter, used for exposition were taken through the entire procedure. A bit width of \( n = 16 \) was used throughout. The iterative error navigation procedure in Section 6.3.3 was performed with 100 iterations, in cases where the procedure did not terminate within this number of iterations, the iteration which had the smallest range of elements in equation (6.8) was returned. In order to demonstrate the properties of the iterative procedure, as well as the starting state being determined by solving the equations (6.7) other starting points were chosen. All the architectures that arose from the multiple starting designs and the results of the performing the iterative refinement were take through logic synthesis, in the same manner as described in the
previous two chapters and the area-delay curves produced.

As part of each iteration, Singular was used to solve a set of multivariate polynomials. The Singular runtime for each iteration for each design was less than a second. The number of variables is the number of nodes in the DFG and in practice it is not expected that the number of nodes will be such that Singular will not run in a reasonable time. The majority of time for an iteration was the Hector runtimes found while performing the flow illustrated in Figure 6.10. For each iteration and each SOP the lemmas were proven serially. There are three SOPs in the cubic polynomial, three in the bilinear interpolation example and four in the Gabor filter, the average runtimes for these proofs for each of these SOPs can be found in Table 6.4. Thus a hundred iterations of the procedure requires orders of hours to run with the Gabor filter requiring approximately 13 hours to finish its iterations. These runtimes can be shrunk by either parallelising the proofs of the lemmas or by constructing the binary array at the same time as constructing the partial product array and thus removing the array creation from being embedded directly into HDL.

6.5.1. **Comparison of Word-length Optimisation versus Array Truncation**

For a given error distribution throughout the DFG, a method of exploiting the error freedom is to use word-length optimisation. For each of the architectures using array truncation techniques an equivalent architecture
Table 6.4.: Runtimes of Hector Lemmas Determining Non Zero Partial Product Bits.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SOP Runtime (s)</th>
<th>SOP Runtime (s)</th>
<th>SOP Runtime (s)</th>
<th>SOP Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cubic Polynomial</td>
<td>40</td>
<td>28</td>
<td>304</td>
<td>–</td>
</tr>
<tr>
<td>Bilinear Interpolation</td>
<td>44</td>
<td>44</td>
<td>53</td>
<td>–</td>
</tr>
<tr>
<td>Gabor Filter</td>
<td>122</td>
<td>108</td>
<td>481</td>
<td>51</td>
</tr>
</tbody>
</table>

was constructed which exploited the same error at each DFG node, but used round towards zero rounding for its error exploitation. For the cases of constant division the prior work found in [141] was used as the only existing fully parallel constant division algorithm. The area-delay graphs below each contain four curves, namely the start and end of the iterative algorithm using array truncation techniques and word-length truncation. The results for the cubic polynomial can be found in Figure 6.11, where the starting set of maximum absolute error bounds for each operator were determining by solving the set of equations (6.7). In addition, other starting error states were considered which gave all the error freedom to some of the nodes. The results from two other starting states can be found in Figures 6.12 and 6.13. These curves demonstrate that the array truncation offers area improvement benefits over the equivalent word-length optimisation, in this case, peaking around 47%. However, Figure 6.12 demonstrates that array truncation, in this case, produces a strictly worse design which is 20% slower and up to 15% larger (note also that the start and end designs of the iterations coincide). In this particular case, part of the word-length design implements the following function:

\[
t_2 = t \times t
\]

\[
t_3 = (t \times t_2) >> r_2
\]

The array truncated version truncates the second of these operators. It has been demonstrated in Chapter 5 that array truncation produces strictly superior implementations when compared to correctly rounded designs, however this property may be negated in the presence of other surrounding operators. Recall from the section on datapath synthesis on page 43 in
Chapter 2 that collections of particular operators can be grouped and optimised as a single datapath operator by logic synthesis tools. One such collection of operators are product-of-sums expressions which include designs of the following form:

\[(A \times B) \times C\]

Such designs are optimised by logic synthesis tools by not performing the final carry propagate addition for the product \(A \times B\), but leaving the result in a redundant form, say \(E+F\). It is then required to implement \((E+F) \times C\), this can be efficiently achieved by creating a Booth type array, more details can be found in [183]. Truncation techniques can be used in addition to such optimisations, however this will require far tighter integration with logic synthesis tools and is the subject of future work. These issues will be the cause of all such anomalies.

![Cubic Polynomial Area-Delay Curves.](image)

Figure 6.11.: Cubic Polynomial Area-Delay Curves.
Figure 6.12.: Cubic Polynomial Area-Delay Curves — Starting State 2.

Figure 6.13.: Cubic Polynomial Area-Delay Curves — Starting State 3.
The result of applying the procedure to the bilinear interpolation can be found in Figure 6.14 and other starting error states in Figures 6.15 and 6.16. These curves demonstrate that the array truncation offers consistent area improvement benefits over the equivalent word-length optimisation, in this case peaking around 44%.

Figure 6.14.: Bilinear Interpolation Area-Delay Curves.
Figure 6.15.: Bilinear Interpolation Area-Delay Curves — Starting State 2.

Figure 6.16.: Bilinear Interpolation Area-Delay Curves — Starting State 3.
The result of applying the procedure to the Gabor filter can be found in Figure 6.17 and other starting states in Figures 6.18 and 6.19. These curves also demonstrate that the array truncation offers consistent area improvement benefits over the equivalent word-length optimisation, in this case peaking around 46%.

Figure 6.17.: Gabor Filter Area-Delay Curves.
Figure 6.18.: Gabor Filter Area-Delay Curves — Starting State 2.

Figure 6.19.: Gabor Filter Area-Delay Curves — Starting State 3.
6.5.2. Validation of Area Heuristic Cost Function

The accuracy of the heuristic area cost function developed in Section 6.4 is now ascertained for the experiments performed. The heuristic area cost function provides a single value which aims to encapsulate the area of the implementation independently of the delay for which the design is synthesised. For each of the nine area-delay graphs in the previous section where, for a given delay, all four designs met timing at that delay the respective areas of each design should ideally correlate with the four associated heuristic area cost values. For each of the nine area-delay graphs in the previous section the delay values for which area figures exist for all four designs were ascertained. Synthesis figures associated with these delays were retained and the rest were discarded. For each graph, at each delay, the designs were given a ranking in terms of their hardware area. These ranks should ideally match the ranks that the area heuristic cost function would order the curves.

For each graph, these ranking pairs were tested using Spearman’s rank correlation coefficient [154] which returns a value of 1 if each of the variables is a perfect monotone function of the other. The correlation coefficients for these nine curves can be found in Table 6.5. The negative correlation coefficient in the case of the cubic polynomial with starting state 2 is related to the anomaly discussed in the previous section where the area heuristic cost function incorrectly asserted that the word-length design will have an implementation area strictly larger than the array truncated version. If this anomalous result is removed from consideration, the average correlation coefficient is 0.91.

The high correlation coefficients lend evidence to the fact that the heuristic area cost function can be used to discriminate between design candidates and thus potentially mitigate the need for costly and time consuming logic synthesis experiments on all candidate designs. However, there is a limitation in the approach used in this heuristic cost function creation. The area heuristic cost function is a sum of area costs for each node, the heuristic does not use the configuration of the nodes in its calculation. In Figure 6.20, node A appears in two locations and the heuristic cost function will assign the same area value to each of these nodes. However, during logic synthesis, the area of the nodes on the critical path will be larger than those
Table 6.5.: Correlation Coefficients for Normalised Area versus Heuristic Area Cost Function.

<table>
<thead>
<tr>
<th>Benchmark Set</th>
<th>Correlation Coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cubic Polynomial</td>
<td>0.95</td>
</tr>
<tr>
<td>Cubic Polynomial Starting State 2</td>
<td>-0.33</td>
</tr>
<tr>
<td>Cubic Polynomial Starting State 3</td>
<td>0.84</td>
</tr>
<tr>
<td>Bilinear Interpolation</td>
<td>0.97</td>
</tr>
<tr>
<td>Bilinear Interpolation Starting State 2</td>
<td>0.89</td>
</tr>
<tr>
<td>Bilinear Interpolation Starting State 3</td>
<td>0.83</td>
</tr>
<tr>
<td>Gabor Filter</td>
<td>0.93</td>
</tr>
<tr>
<td>Gabor Filter Starting State 2</td>
<td>1.00</td>
</tr>
<tr>
<td>Gabor Filter Starting State 3</td>
<td>0.85</td>
</tr>
</tbody>
</table>

off the critical path. Given that there is one A node on and off the critical path, their actual hardware area costs will differ, despite the area heuristic cost function assigning the same area cost to both. It is this context insensitivity that potentially undermines the applicability of the area heuristic cost function created. Extending the heuristic cost function to overcome this limitation is the subject of future work.

Figure 6.20.: Sample DFG Highlighting a Limitation of the Area Heuristic Cost Function.

6.5.3. Quality of Error Landscape Navigation

In order to determine the quality of the iterative procedure put forward in Section 6.3.3, all the start and end designs associated with the cubic...
polynomial were extracted from Section 6.5.1 and combined in a single figure, Figure 6.21 (note that the iteration using starting state 1 returned an end state identical to the start state and has thus been omitted from this figure). Figure 6.21 shows that the iterative procedure results in implementations which are typically 5% and up to 18% larger when compared to the implementation associated with the starting state. However, repeating the exercise for the bilinear interpolation and the Gabor filter results in Figures 6.22 and 6.23 respectively (a zoom of Figure 6.23 can be found in Figure 6.24). These figures demonstrate a consistent advantage of performing the iterative procedure, with an average area improvement of 11% and 23% for the bilinear interpolation and Gabor filters respectively and a maximum percentage area improvements are 40% and 87% respectively. The advantage seen by the application of the iterative procedure will depend upon the proximity of the starting state to a local extremum. Certainly, if the starting state has a global error that does not fully exploit the global error freedom, the iterative procedure will certainly produce error allocations which do fully exploit the freedom.

Figure 6.21.: Area-Delay Curves for the Start and End Implementations of the Iterative Procedure for the Cubic Polynomial.
Figure 6.22.: Area-Delay Curves for the Start and End Implementations of the Iterative Procedure for the Bilinear Interpolation.

Figure 6.23.: Area-Delay Curves for the Start and End Implementations of the Iterative Procedure for the Gabor Filter.
Figure 6.24.: Zoom of Area-Delay Curves for the Start and End Implementations of the Iterative Procedure for the Gabor Filter.

The iterative procedure has the potential to refine a given error distribution and return another whose exploitation results in implementations with reduced hardware implementation costs. However, the iterative procedure returns a design which satisfies the optimality condition in equation (6.2) on page 194, which is the condition for an extremum of the error landscape. We are unable to establish an a priori method for determining the nature of this extremum, namely whether it is a local maximum or minimum.

The start state which uses the default error distribution defined by solving the set of equations (6.7) and the result of performing the iterative procedure on this collection of errors are both strong design choices. To illustrate this, these two implementations were synthesised along with implementations which had arbitrary error assignments to each node of the respective DFG which also met the global absolute error requirement. The results for the cubic polynomial, bilinear interpolation and Gabor filter can be found in Figures 6.25, 6.26 and 6.27 respectively. These results demonstrate an average area percentage benefit of 19%, 24% and 22% for the three design respectively and an area percentage benefit which peaks at 51%, 41% and 38% respectively.
Figure 6.25.: Area-Delay Curves for the Arbitrary, Start and End Implementations for the Cubic Polynomial.

Figure 6.26.: Area-Delay Curves for the Arbitrary, Start and End Implementations for the Bilinear Interpolation.
6.5.4. Conclusion

In conclusion, this chapter has demonstrated how to exploit arbitrary error specifications within a DFG consisting of sum-of-product and constant division operators, as would occur in a given architecture for implementing a fixed-point polynomial with rational coefficients. A novel approach to navigating the allocation of errors within the DFG, which attempts to move towards extrema, has been presented. Experimental results showed that the new approach to exploiting errors demonstrates consistent benefits over word-length optimisation with example area reduction up to 46%. The iterative approach to navigating the error landscape moves towards extrema, in the cases where the approach moves towards a minimum, the area reduction benefits can be up to 87%. The approach does offer a consistent advantage over an arbitrary error assignment which meets the global maximum absolute error bound with an area benefit up to 51% in experiments performed. In addition, an area heuristic cost function has been put forward which has been shown to have a high Spearman’s rank correlation coefficient with respect to actual logic synthesis experiments and can be used, with high confidence, in correctly discriminating between design alternatives and can
thus be used as part of the iterative procedure in discounting potentially suboptimal designs.
7. Lossy Floating-Point Polynomial Synthesis

For the final technical chapter, we turn to the challenge of accurately implementing polynomials with floating-point arithmetic. In doing so, we leverage some of the contributions from Chapter 5. Floating-point arithmetic provides the greater dynamic range which fixed-point arithmetic lacks. However, floating-point implementation costs are orders of magnitude greater than fixed-point designs. In applications where the range of inputs is not known at design time, such as radar for navigation, a wide dynamic range is required. Increased dynamic range can also be required as a result of the algorithm itself, for example it make be required to implementation operations such as exponentiation. Finally there are naturally occurring input data types which require high fidelity around zero but limited precision for large values, such as audio signals, these naturally lend themselves to the use of floating-point numbers [61]. High performance implementations for such applications necessitates the creation of hardware that implements floating-point arithmetic. Errors introduced by floating-point computation are relative in nature and therefore it is natural to seek floating-point polynomial implementations with a bounded relative error. If \( \hat{p} \) is the result of an implementation attempting to calculate \( p \), the relative error in doing so is:

\[
\frac{\hat{p} - p}{p}
\]

Correctly determining the sign of a polynomial requires the relative error to be strictly less than 1. If a relative error of greater than or equal to 1 were permitted, say:

\[
\frac{\hat{p} - p}{p} \leq 1 + \varepsilon \quad \varepsilon \geq 0
\]
Then returning $\hat{p} = -\varepsilon p$ would be permissible but $p$ and $\hat{p}$ would differ in their sign. There are countless algorithms in computer aided geometric design which involve intersections, distance functions and curvature extrema polynomial calculations \cite{78}, which require correctly determining the sign of a polynomial. Surfaces can be tangential or singular and errors in polynomial evaluation can result in gross errors or catastrophic failure in geometric calculation \cite{77}. In such applications the smallest values still need an evaluation with correct leading digits and zero, when it occurs, needs to be exact. If an implementation exhibits a relative error of strictly less than 1, then the algorithm is said to \textit{accurately evaluate} the polynomial in question.

Given the implementation cost of floating-point operations, the use of increased precision to gain greater accuracy can degrade performance in both time and power consumption. Floating-point polynomial implementations therefore need to be relatively accurate and only use the appropriate level of precision for evaluation. This chapter provides background on floating-point arithmetic, a survey of floating-point polynomial implementation techniques as well as our construction of floating-point multipliers with bounded relative error. The state of the art in terms of accurate evaluation of polynomials is a non-constructive framework, the main contribution of this chapter is the presentation of an algorithmic construction which uses the appropriate precision for different regions of the input domain as well as steps towards the automation of the technique.

7.1. Floating-Point Background and Algorithms for Implementing Polynomials

Floating-point numbers, as defined in the IEEE-754 floating-point standard \cite{1}, are represented by the triple: sign, exponent and mantissa, $(s, \exp, \text{mant})$. Excluding exceptional cases, these numbers are interpreted as, for a fixed integer \textit{bias}:

\[ (-1)^s 2^{\exp - \text{bias}} 1.\text{mant} \]

where juxtaposition denotes multiplication and the point is a binary point in a fixed point number. The binary width of the exponent and mantissa, \textit{ew} and \textit{mw} respectively for the various standard types of floating-point
Table 7.1.: Floating-Point Types.

<table>
<thead>
<tr>
<th>Type</th>
<th>Notation</th>
<th>Sign Width</th>
<th>Exponent Width (ew)</th>
<th>Mantissa Width (mw)</th>
<th>Bias</th>
<th>Roundoff Error (u)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Half</td>
<td>F16</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>$2^{-11}$</td>
</tr>
<tr>
<td>Single</td>
<td>F32</td>
<td>1</td>
<td>8</td>
<td>23</td>
<td>127</td>
<td>$2^{-24}$</td>
</tr>
<tr>
<td>Double</td>
<td>F64</td>
<td>1</td>
<td>11</td>
<td>52</td>
<td>1023</td>
<td>$2^{-53}$</td>
</tr>
<tr>
<td>Quad</td>
<td>F128</td>
<td>1</td>
<td>15</td>
<td>112</td>
<td>16383</td>
<td>$2^{-113}$</td>
</tr>
</tbody>
</table>

Table 7.2.: Floating-Point Interpretation.

<table>
<thead>
<tr>
<th>sign</th>
<th>exponent exp</th>
<th>mantissa</th>
<th>Value</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 1</td>
<td>0</td>
<td>0</td>
<td>$(−1)^{s}0$</td>
<td>± Zero</td>
</tr>
<tr>
<td>0, 1</td>
<td>0</td>
<td>$\neq 0$</td>
<td>$(-1)^{s}2^{1-bias}0.mant$</td>
<td>Denormal</td>
</tr>
<tr>
<td>0, 1</td>
<td>$0 &lt; exp &lt; 2^{ew} - 1$</td>
<td>mant</td>
<td>$(-1)^{s}2^{exp−bias}1.mant$</td>
<td>Normal Numbers</td>
</tr>
<tr>
<td>0, 1</td>
<td>111...111</td>
<td>0</td>
<td>$\pm\infty$</td>
<td>± Infinity</td>
</tr>
<tr>
<td>0, 1</td>
<td>111...111</td>
<td>$\neq 0$</td>
<td>Not a Number</td>
<td>NaN</td>
</tr>
<tr>
<td>0, 1</td>
<td>111...110</td>
<td>111...111</td>
<td>$(−1)^{s}2^{bias}(2−2^{−mw})$</td>
<td>± MAX_FLOAT</td>
</tr>
</tbody>
</table>

numbers can be found in Table 7.1. The number of bits of precision or significant bits is $mw + 1$ as the floating-point format has an implied one. The roundoff error $u$ is half the distance between 1 and the next representable floating-point value.

The particular interpretations of the floating-point numbers can be found in Table 7.2.

Despite having a large dynamic range and representing large integers and very small fractional numbers, any appearance that floating-point numbers represent the real numbers is purely illusory. Key properties of real numbers are routinely violated by floating-point arithmetic. For example (denoting $\hat{+}$, $\hat{−}$ and $\hat{\times}$ as floating-point operations) there exist $x$, $a$, $b$ and $c$ such that:

$$x\hat{+}(+0) \neq x \quad \text{e.g.} \quad (−0)\hat{+}(+0) = +0$$

$$(a\hat{+}b)\hat{+}c \neq a\hat{+}(b\hat{+}c)$$

$$(a\hat{\times}b)\hat{\times}c \neq a\hat{\times}(b\hat{\times}c)$$

$$a\hat{\times}(b\hat{+}c) \neq (a\hat{\times}b)\hat{+}(a\hat{\times}c)$$

$$a\hat{+}b = a \quad \not\Rightarrow \quad b = 0$$

242
Accurate computation in light of these fundamental difficulties has given rise to a plethora of algorithms, for example error-free transformations. These background algorithms involve floating-point operations which are assumed to be correctly rounded, in that each floating-point operation will return the nearest floating-point number using RTE rounding.

7.1.1. Error-Free Transformations

It is known that for the basic floating-point operations $+,-,\times$ the error introduced is representable as a floating-point number. The following floating-point algorithms return this representable error.

The Two Sum algorithm for addition [100]:

$$TwoSum(a, b) = [x, y]$$

$$x = a \hat{+} b$$

$$z = x \hat{-} a$$

$$y = (a \hat{-} (x \hat{-} z)) \hat{+} (b \hat{-} z)$$

In order to do error free multiplication floating-point inputs need splitting into two floating-point numbers with half the number of significant bits. If the floating-point format has $p$ significant bits then a split can be achieved via [44]:

$$Split(a) = [x, y]$$

$$factor = 2^{\lceil p/2 \rceil} + 1$$

$$c = factor \times a$$

$$x = c \hat{-} (c \hat{-} a)$$

$$y = a \hat{-} x$$
Having performed this split then error free multiplication is:

$$TwoProduct(a, b) = [x, y]$$

$$x = a \hat{\times} b$$

$$[a_1, a_2] = Split(a)$$

$$[b_1, b_2] = Split(b)$$

$$y = (a_2 \hat{\times} b_2)^\hat{-}((x\hat{-}(a_1 \hat{\times} b_1))^\hat{-}(a_2 \hat{\times} b_1))^\hat{-}(a_1 \hat{\times} b_2))$$

7.1.2. Polynomial Evaluation

There are a variety of strategies to implement a polynomial. Consider a univariate polynomial of the following form:

$$p(x) = \sum_{i=0}^{n} a_i x^i$$

The dot product implementation is to form the power of $x$ by repeated multiplication and perform the final coefficient multiplication and summation as a dot product:

$$t_0 = a_0$$

$$y_0 = 1$$

for $i = 1..n$

$$y_i = y_{i-1} \hat{\times} x$$

$$z_i = y_i \hat{\times} a_i$$

$$t_i = t_{i-1} + z_i$$

end

$$DotProduct(p) = t_n$$

In [144], an absolute error bound for the dot product implementation was found. This is an a posteriori bound, in that it is a function of $DotProduct(p)$, so no a-priori bound was presented. The analysis assumed $|x| \leq 2^{-k}$ for some integer $k \geq 1$, $|a_i| \leq |a_0| = 1$ and that no denormalised numbers were produced during the computation.

A classic method for implementing a univariate polynomial is by using a
Horner expansion:

\[
\begin{align*}
  s_n &= a_n \\
  \text{for } i &= n-1..0 \\
  s_i &= (s_{i+1} \times x) + a_i \\
\end{align*}
\]

\[
\text{end}
\]

\[\text{Horner}(p) = s_0\]

The relative error can be bounded as follows, where \(u\) is the unit roundoff

\[
\left| \frac{\text{Horner}(p) - p}{p} \right| \leq nu \sum_{i=0}^{n} |a_i||x|^i \left(1 - nu\right) |p|
\]

This classic approach can be enhanced by using the error free transformations giving rise to the compensated Horner scheme [105] [70]:

\[
\begin{align*}
  s_n &= a_n \\
  s'_n &= 0 \\
  \text{for } i &= n-1..0 \\
  [p_i, tp_i] &= \text{TwoProduct}(s_{i+1}, x) \\
  [s_i, ts_i] &= \text{TwoSum}(p_i, a_i) \\
  s'_i &= (s'_{i+1} \times x) + (tp_i + ts_i) \\
\end{align*}
\]

\[
\text{end}
\]

\[\text{CompHorner}(p) = s_0 + s'_0\]

The relative error can be bounded as follows [105]:

\[
\left| \frac{\text{CompHorner}(p) - p}{p} \right| \leq u + \frac{4n^2u^2 \sum_{i=0}^{n} |a_i||x|^i}{\left(1 - nu\right)^2 |p|}
\]

Note that in both cases if \(\sum_{i=0}^{n} |a_i||x|^i\) is significantly greater than \(|p|\) then there is no guarantee of small relative error. This approach essentially extracts the error at each floating-point operation within the Horner algorithm and evaluates another polynomial whose coefficients are these errors:

\[\text{CompHorner}(p) = \text{Horner}(p) + \text{Horner}(\bar{p})\]

245
where $\tilde{p}$ has coefficients $tp_i + ts_i$. In [71], this approach is extended to compute $p$ has a sum of polynomials evaluated in a Horner fashion called K-times compensated Horner.

In [160] a polynomial close to the original is evaluated in its place, which is chosen be have an exact evaluation in the Horner scheme; the method is not guaranteed to return better accuracy than Horner but did exhibit a typical improvement of between a 100 and 1000 factor of accuracy improvement over Horner. In [41], it is noted that extreme accuracy degradation can occur due to catastrophic cancellation. For example, $2^{100} - 2^{100} + 1$ will return 0 or 1 when using single precision floating-point arithmetic, depending on the order the additions are performed. This is due to complete cancellation of the dominant terms $2^{100}$ and $-2^{100}$. The approach in [41] attempts to set coefficients to zero that would otherwise cause catastrophic cancellation in a Horner evaluation scheme. This work is targeted at univariate polynomials for function evaluation. Iterative methods of polynomial evaluation have also been considered in [78] which require solving a system of linear equations, in certain situations this method will fail to converge.

The dot product and Horner forms of polynomial implementation are two of many potential implementations. For example the number of distinct ways of summing $n$ inputs with a two input adder is $(2n - 1)!!$ and the number of ways of creating $x^n$ with a two input multiplier is the $n$th Wedderburn-Etherington number [130] [53], which asymptotically approaches:

$$\frac{0.31877 \times 2.48325^n}{n^{3/2}}$$

Suffice to say, there are a myriad of potential rewritings of polynomials in terms of two input operators. Each one of these rewritings exposes a different level of potential parallelism opportunities, operator count and accuracy. Navigating this space is the subject of [115] where intermediate variables are represented as intervals and are augmented with a notion of quality, for example the distance between the infinitely precise and floating-point value. These quality values are statically propagated throughout the operations in such a way that operators that introduce the greatest inaccuracy can be identified and subsequently modified by using rewrite rules such as associativity, symmetry and distributivity. In this way, floating-point implementations can have their accuracy improved. The factorial set
Table 7.3.: Accurate Evaluability of Polynomials.

<table>
<thead>
<tr>
<th>Polynomial</th>
<th>Accurately Evaluable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x + y + z$</td>
<td>No</td>
</tr>
<tr>
<td>$(a - b)(c - d)(e - f)$</td>
<td>Yes</td>
</tr>
<tr>
<td>$z^6 + x^2 y^2 (x^2 + y^2 - 3z^2)$</td>
<td>Yes</td>
</tr>
<tr>
<td>$z^6 + x^2 y^2 (x^2 + y^2 - 4z^2)$</td>
<td>No</td>
</tr>
</tbody>
</table>

of potential rewrites has been expressed in [84] in a polynomial fashion using an Abstract Program Equivalence Graph.

All of these techniques either offer no accuracy guarantee or give an error bound which is unacceptability large when the polynomial in question is close to zero. In [114], it is noted that relative error can be infinite when the polynomial’s true value is zero, factorising the polynomial can solve this problem but it can be hard to find the roots of polynomials to high enough accuracy. The approach that does offer accurate evaluation as well as the potential of only using extra precision where necessary is that put forward by Demmel et al. in [45], [47] and [46]. True lossy synthesis requires the complete control of the accuracy involved, as such this is the approach taken here.

The results in [46] contain the surprising result that it is not possible to accurately evaluate certain polynomials with only floating-point multiplication and addition operators. Examples of polynomials that can and cannot be accurately evaluated can be found in Table 7.3. These results are derived by consideration of the affine variety of the polynomials in question. The definition of the affine variety, hereafter called the variety, is as follows:

**Definition** Affine Variety:

$$V(f_1, f_2, ..., f_s) = \{(a_1, a_2, ..., a_n) \in k^n : f_i(a_1, a_2, ..., a_n) = 0, i = 1, 2, ..., s\}$$

The variety is the set of inputs which return zero when evaluated for all $f_i$. A theorem is stated and proven in [46] which gives a necessary condition on the form of the variety of a polynomial in order for an accurate evaluation to exist. It is also observed that as the variety is approached from different directions, the behaviour of the polynomial is determined by particular dominant terms. A method for determining the directions and the dominant terms is presented and that it is proven that if each of these
dominant terms can be accurately evaluated then the entire polynomial can be accurately evaluated. A proof is also given that a strictly positive homogeneous polynomial can be accurately evaluated. These results provide a framework within which an irreducible homogeneous polynomial can be accurately evaluated. However these results were non-constructive in nature and an algorithm and error analysis were not presented. We further this work by presenting an algorithmic construction by way of finalising one of the primary examples used in [46], namely that of the Motzkin polynomial to show that it works in practice. During the construction of this worked example, the implicit power of the method put forward in [46] to allow the use of only just enough precision is made explicit.

The rest of this chapter begins with the construction of floating-point multipliers and adders. Given that the goal is polynomial evaluation with a bounded relative error then the components upon which the evaluation is based need only bounded relative error and need not be correctly rounded. We present the construction of lossy floating-point multipliers and adders which exploit the freedom provided by bounded relative error to create components with reduced hardware implementation costs. This is followed by Section 7.3 which introduces the idea of an allowable variety and gives a necessary condition for polynomial accurate evaluation contained within [46]. This is then followed by Section 7.4 containing the complete Motzkin polynomial worked example. This section is highly analytic and the associated appendices are very specific but crucial to the contribution of this chapter. The chapter closes with the challenges and steps to generalisation of the technique in Section 7.5.
7.2. Lossy Floating-Point Components

Implementing polynomials with bounded relative error accuracy requires floating-point components with bounded relative error accuracy. If the floating-point components can be made to be faithfully rounded then they will have a relative error of $2^{-mw}$ where $mw$ is the output mantissa width. If the infinitely precise answer is representable then this answer will be returned by a faithfully rounded floating-point operation and will thus have zero relative error. Otherwise the infinitely precise answer (assumed to be of the form $(-1)^s2^e1.m$) resides between two representable numbers (where $m1$ and $m2$ have $mw$ bits of precision):

$$(-1)^s2^e1.m1 < (-1)^s2^e1.m < (-1)^s2^e1.m2$$

where $|m2 - m1| = 2^{-mw}$

The relative error in returning either of these two neighbouring floating-point numbers is:

$$\left|\frac{(-1)^s2^e1.m1 - (-1)^s2^e1.m}{(-1)^s2^e1.m}\right| = \frac{1.m1 - 1.m}{1.m} < 2^{-mw}$$

$$\left|\frac{(-1)^s2^e1.m2 - (-1)^s2^e1.m}{(-1)^s2^e1.m}\right| = \frac{1.m2 - 1.m}{1.m} < 2^{-mw}$$

So it suffices to make faithfully rounded floating-point components as these will have a relative error of strictly less than $2^{-mw}$. A technique for constructing faithfully rounded floating-point adders and their hardware implementation benefits can be found in Appendix B. Faithfully rounded integer multipliers can be leveraged in constructed floating-point multipliers. We now show, as has already been shown in previous chapters, that relaxing correct rounded for faithful rounding gives rise to the opportunity to improve upon the hardware implementation costs.

7.2.1. Lossy Floating-Point Multipliers

We now consider how to construct faithfully rounded floating-point multipliers whose inputs are assumed to exclude denormal and exceptional cases and are represented by the triples $(sa, expa, manta)$ and $(sb, expb, mantb)$
and return an output in the form \((sy, expy, manty)\). The equations governing the outputs are:

\[
y \approx a \times b \\
(-1)^{sy} 2^{expy - bias} 1.my \approx (-1)^{sa} 2^{expa - bias} 1.ma \times (-1)^{sb} 2^{expb - bias} 1.mb \\
(-1)^{sy} 2^{expy - bias} 1.my \approx (-1)^{sa \oplus sb} 2^{(expa + expb - bias) - bias} (1.ma \times 1.mb)
\]

\[
sy = sa \oplus sb \\
expy = expa + expb - bias \\
1.my = 1.ma \times 1.mb
\]

These equations need slight modification given that \(1.ma \times 1.mb\) produces numbers in the interval \([1,4)\) and so a one bit renormalisation may be required as well as rounding. The fixed point steps to producing \(my\), for \(mw\) bit mantissas, are thus:

\[
a[mw : 0] = 2^{mw} + ma // adding in the implied one \\
b[mw : 0] = 2^{mw} + mb \\
c[m - 1 : 0] = multFR(a, b) \\
my = (c[m - 1] == 1)? c[m - 2 : m - mw - 1] : c[m - 3 : m - mw - 2]
\]

where \(multFR\) returns a faithful rounding of the top \(m\) bits of the multiplication of \(a\) and \(b\). Now \(multFR\) can be any of the truncation schemes constructed in Chapter 5. In order to construct the most hardware efficient floating-point multiplier, a design with the smallest precision for the intermediate variable \(c\) is desirable. What is the smallest value of \(m\) such that the floating-point multiplier is faithfully rounded?

### 7.2.2. Faithfully Rounded Floating-Point Multiplier Theorem

We now state and prove our theorem regarding a sufficient bit width for the internal variable \(c\) in order for the floating-point multiplier to be faithfully rounded.
Theorem 7.2.1 Faithfully Rounded Floating-Point Multiplier Theorem —
A floating-point multiplier with input and output mantissa of width mw
is guaranteed to be faithfully rounded if the integer multiplier is faithfully
rounded keeping mw + 2 bits of precision.

The proof splits into the following cases:

- **Case:** $c[mw + 1] = 0$. In this case $my = c[mw − 1 : 0]$ which is
faithfully rounded due to definition of $\text{multFR}$, hence in this case the
floating-point multiplier is faithfully rounded.

- **Case:** $c[mw + 1] = 1$ and $c[0] = 0$. If $c$ is a fixed point number
of the form $2.mw$ and the infinitely precise answer is $r$, then during
renormalisation, $c[0]$ is removed. Hence $r$ and $c$ are related as follows:

$$|r - c| < 2^{-mw} < 2^{-mw+1}$$

$$|r - c| < 2^{-mw+1}$$

In this case, one unit in the last place is $2^{-mw+1}$, hence this meets the
accuracy condition.

- **Case:** $c[mw + 1] = 1$ and $c[0] = 1$. Then from the definition of
$\text{multFR}$

$$|r - c| < 2^{-mw}$$

$$-2^{-mw} < r - c < 2^{-mw}$$

$$0 < r - (c - 2^{-mw}) < 2^{-mw+1}$$

$$|r - (c - 2^{-mw})| < 2^{-mw+1}$$

In this case one unit in the last place is $2^{-mw+1}$. Also, due to renormalisation, the answer returned is $c - 2^{-mw}$. Due to this inequality, it can be see that the result will be within one unit in the last place and hence faithfully rounded.
In conclusion, if the following fixed-point algorithm is used as part of a floating-point multiplier, the entire design will be faithfully rounded:

\[
\begin{align*}
a[mw : 0] &= 2^{mw} + ma \\
b[mw : 0] &= 2^{mw} + mb \\
c[mw + 1 : 0] &= \text{multFR}(a, b) \\
my &= (c[mw + 1] == 1)? c[mw : 1] : c[mw - 1 : 0]
\end{align*}
\]

To see the benefits of our faithfully rounded floating-point multipliers, these were synthesised against reference designs from Synopsys DesignWare library [163] for various rounding modes. Comparisons were performed for multipliers with F16 and F32 inputs. The results are illustrated in Figures 7.1 and 7.2. These figures contain DesignWare round to nearest, ties to even (DesignWare RTE) and round towards zero (DesignWare RTZ) adders and the faithfully rounded architecture (Proposed Faithfully Rounded). The results show that F16 faithfully rounded multipliers can be 9% faster than DesignWare and up to 54% smaller. The results show that F32 faithfully rounded multipliers can be 4% faster than DesignWare and up to 45% smaller.
Figure 7.1.: Area-Delay Curves for F16 Multiplier Architectures.

Figure 7.2.: Area Delay Curves for F32 Multiplier Architectures.
7.3. Accurate Evaluation and Allowable Varieties

Having established how to construct floating-point multipliers and adders with bounded relative error, we now turn to how to use these components to implement a polynomial with bounded relative error. The first key idea from the paper by Demmel et al. [46] is that of an allowable variety. This section presents a more informal demonstration of the notion, a detailed proof of these concepts can be found within the paper itself. Accurate evaluation requires that the varieties of both the polynomial \( p \) and its implementation \( \hat{p} \) are identical. Otherwise the following situations can occur:

\[
p = 0 \neq \hat{p} \implies \left| \frac{\hat{p} - p}{p} \right| = \left| \frac{\hat{p}}{0} \right| = \infty
\]

\[
\hat{p} = 0 \neq p \implies \left| \frac{\hat{p} - p}{p} \right| = \left| \frac{p}{p} \right| = 1
\]

Both situations result in an unacceptable relative error. So a necessary condition for accurate evaluation is that the varieties of \( p \) and \( \hat{p} \) are identical. It is now assumed that only the floating-point operators of addition and multiplication are available and that these operators have the following property:

\[
\hat{a} + \hat{b} = 0 \implies a + b = 0
\]

\[
\hat{a} \times \hat{b} = 0 \implies ab = 0
\]

These conditions hold if no underflow or overflow occur. Despite round off error, it is possible that the varieties of \( p \) and \( \hat{p} \) are identical, for example consider the following polynomial:

\[
p = (a - b)(c - d)(e - f)
\]

\[
t_0 = (a\hat{\hat{b}})
\]

\[
t_1 = (c\hat{\hat{d}})
\]

\[
t_2 = (e\hat{\hat{f}})
\]

\[
t_3 = t_0 \times t_1
\]

\[
\hat{p} = t_3 \times t_2
\]
If $p = 0$ then $a = b$ or $c = d$ or $e = f$ which then implies that $\hat{p} = 0$. Conversely:

\[
\begin{align*}
\hat{p} &= 0 \\
\Rightarrow & \quad t_3 = 0 \quad \text{or} \quad t_2 = 0 \\
\Rightarrow & \quad t_0 = 0 \quad \text{or} \quad t_1 = 0 \quad \text{or} \quad e = f \\
\Rightarrow & \quad a = b \quad \text{or} \quad c = d \quad \text{or} \quad e = f \\
\Rightarrow & \quad p = 0
\end{align*}
\]

Thus, in this case, the variety of $p$ and $\hat{p}$ are identical, so there is potential that there exists an accurate evaluation of $p$. However, this simple polynomial fails the condition:

\[
\hat{p} = (a \hat{+} b) \hat{+} c
\]

In this case, consider $c = -1$. There are are many floating-point numbers $a$ and $b$ whose output from a floating-point adder is 1 but whose infinitely precise sum is not 1. This means that $p \neq 0 = \hat{p}$ and the relative error is unacceptable. In the general case, consider the situation where a floating-point implementation outputs 0 and there exists an intermediate signal within the implementation, say $X \neq 0$, which is the output of operator $Y$. The inputs to $Y$ can be altered while leaving output $X$ unchanged and hence the entire implementation still returns 0. Thus it is possible that $\hat{p} = 0$ but the infinitely precise result $p \neq 0$. (There is a caveat that changing $X$ must alter the output of the implementation for this argument to hold, \textit{i.e.} it should not be subsequently multiplied by 0. In this sense, $Y$ must be a non-trivial operator. Obviously when the implementation returns 0, there must exist some non-trivial operators, ignoring the trivial case when the polynomial to be implemented is a constant 0.) Thus, a necessary condition for there to exist an accurate evaluation of $p$ is that there exists an implementation which when it returns 0 has all non-trivial operators producing 0 within it. Some of the non-trivial operators act on the inputs to the polynomial, if these are producing 0, then certain inputs must have the property that $x_j = 0$ or $x_j \pm x_k = 0$. Conclude that if an accurate evaluation of $p$ exists whose output is $\hat{p}$, then $\hat{p}$ has a variety which is the union of constraints of the form $x_i = 0$ or $x_j \pm x_k = 0$. Moreover, given that the varieties of $p$ and
Table 7.4.: Allowable Varieties and Accurate Evaluability of Polynomials.

<table>
<thead>
<tr>
<th>Polynomial</th>
<th>Variety</th>
<th>Allowable Variety?</th>
<th>Accurately Evaluable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x + y + z$</td>
<td>$x + y + z = 0$</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>$(a - b)(c - d)(e - f)$</td>
<td>$a - b, c - d, e - f = 0$</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>$z^5 + x^2 y^2 (x^2 + y^2 - 3z^2)$</td>
<td>$z = y = 0, z = x = 0,$</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>$\pm x = \pm y = \pm z$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$z^5 + x^2 y^2 (x^2 + y^2 - 4z^2)$</td>
<td>$\ldots, x + y + z = 0, \ldots$</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

$\hat{p}$ must be identical if $\hat{p}$ is an accurate evaluation of $p$, then the variety of $p$ is the union of constraints of the form $x_i = 0$ or $x_j \pm x_k = 0$. If $p$ has this property, then its variety is said to *allowable* and *unallowable* otherwise. This notion of allowability provides the rationale for the results found in Table 7.3 on page 247. The varieties and their allowability of the polynomials found in Table 7.3 can be found in Table 7.4. Note that the definition of allowability is linked to the fact that only floating-point addition and multiplication are the operators being considered.
7.4. Accurate Evaluation of the Motzkin Polynomial

We now supplement the framework put forward in [46] with an algorithmic procedure and complete error analysis in order to establish an implementation which accurately evaluates the Motzkin polynomial. The framework focuses on homogeneous irreducible polynomials of which the Motzkin polynomial is an example. This is an accurate evaluation of the Motzkin polynomial where the inputs are assumed to be F16 non exceptional (only zero and non denormalised numbers are assumed to be valid inputs) and the required output relative accuracy is $2^{-10}$ in keeping with a faithful rounding to an F16 output. This particular instance is chosen to be complex enough to stress the validity of the method, but small enough for the result to be extensively verified. The aim is to use the standard floating-point types, F16, F32, F64 and F128 only when necessary and these operations are faithfully rounded, such that if an operator has output width $mw$ it will have a relative error of strictly less than $2^{-mw}$.

The Motzkin polynomial [125], was first introduced in 1967 by Theodore Motzkin in response to Hilbert’s non constructive 1888 proof that there exist polynomials which are non-negative but cannot be written as a sum of squares of polynomials, and is a primary example in [46]. This polynomial is defined as follows:

$$p(x, y, z) = z^6 + x^2 y^2 (x^2 + y^2 - 3z^2)$$

Motzkin proved that this polynomial is non negative by use of the arithmetic-geometric mean property which can be proven using Jensen’s inequality [88], that for any non-negative numbers $x_1, x_2, ..., x_n$:

$$\frac{x_1 + x_2 + ... + x_n}{n} \geq (x_1 x_2 \ldots x_n)^{1/n}$$

With equality occurring when $x_1 = x_2 = ... = x_n$. Substituting $n = 3$, $x_1 = z^6$, $x_2 = y^4 x^2$ and $x_3 = y^2 x^4$ then the following equality is derived for
any real $x, y$ and $z$:

$$\frac{z^6 + y^4x^2 + y^2x^4}{3} \geq (x^6y^6z^6)^{1/3} = z^2y^2x^2$$

$$\Rightarrow z^6 + x^2y^2(x^2 + y^2 - 3z^2) \geq 0$$

With equality occurring when $z^6 = y^4x^2 = y^2x^4$. So the variety of the Motzkin polynomial is:

$$z^6 = y^4x^2 = y^2x^4$$
$$\pm z^3 = \pm y^2x = \pm yx^2$$
$$z = x = 0 \quad \text{or} \quad z = y = 0 \quad \text{or} \quad \pm z = \pm y = \pm x$$

The six lines that constitute the variety are represented in Figure 7.3. Note that the variety is *allowable* and hence an accurate evaluation is potentially
possible, using only floating-point adder and multiplier components.

7.4.1. Partitioning the Domain

Accurate evaluation near these lines will require the greatest amount of precision. Far from these lines a lesser precision is possible. The level of precision will depend on the proximity to one of these lines. Hence, in order to ascertain the level of precision required the input domain must be partitioned into regions which are closer to one line of the variety as opposed to another. Partitioning the line $x = y = z$ from the others can be performed on each 2D projection, the projection of the lines of the variety onto the $xy$ plane can be seen in Figure 7.4. It is required to create the constraints such that an arbitrary point $(x, y)$ is closer to the line $x = y$.

Figure 7.4.: Projection of the Variety of the Motzkin Polynomial onto the $xy$ Plane.
than the other lines $x = 0$, $y = 0$ or $x = -y$. This can be rephrased as:

$$(x, y) \text{ must be closer to } x - y = 0 \text{ than } x = 0 \ y = 0 \ x + y = 0$$

$$\Rightarrow |x - y| \leq |x| \ |x - y| \leq |y| \ |x - y| \leq |x + y|$$

$$\Rightarrow |x - y| \leq |x| \ |x - y| \leq |y| \ sgn(x) = sgn(y)$$

where $sgn(x)$ returns the sign of $x$, -1 if $x < 0$, 1 if $x > 0$ and 0 if $x = 0$. The projections onto the other planes can be found in Figures 7.5 and 7.6.

Combining the results of creating the constraints that partition the line

![Graph](image)

**Figure 7.5:** Projection of the Variety of the Motzkin Polynomial onto the $yz$ Plane.
Figure 7.6.: Projection of the Variety of the Motzkin Polynomial onto the $xz$ Plane.

$x = y = z$ from the others in each projection together gives:

$$sgn(x) = sgn(y) = sgn(z)$$
$$|x - y| \leq |x|$$
$$|x - y| \leq |y|$$
$$|x - z| \leq |z|$$
$$|y - z| \leq |z|$$
Repeating the exercise for the lines $z = x = 0$ and $z = y = 0$ and summarising, there are three subdomains:

Subdomain for $x = y = z$

$$\text{sgn}(x) = \text{sgn}(y) = \text{sgn}(z)$$

$$|x - y| \leq |x| \quad |x - y| \leq |y| \quad |x - z| \leq |z| \quad |y - z| \leq |z|$$

Subdomain for $z = x = 0$

$$|x| \leq |x - y| \quad |x| \leq |x + y| \quad |x| \leq |y| \quad |z| \leq |y - z| \quad |z| \leq |y + z|$$

Subdomain for $z = y = 0$

$$|y| \leq |y - x| \quad |y| \leq |y + x| \quad |y| \leq |x| \quad |z| \leq |x - z| \quad |z| \leq |x + z|$$

At this point it is noted that the Motzkin polynomial is unchanged when any of $x$, $y$ or $z$ are negated, so it suffices to accurately evaluate the polynomial over the positive octant, namely $x, y, z \geq 0$. In this octant the subdomain descriptions can be simplified to:

Subdomain for $x = y = z$

$$x \leq 2y \quad y \leq 2x \quad x \leq 2z \quad y \leq 2z$$

Subdomain for $z = x = 0$

$$2x \leq y \quad 2z \leq y$$

Subdomain for $z = y = 0$

$$2y \leq x \quad 2z \leq x$$

Note that the conditions which define the subdomains can be computed without incurring floating-point error as they include multiplications by 2 and comparisons.

**7.4.2. Accurate Evaluation Far from the Variety**

In each subdomain, far from the line at which $p = 0$, the Motzkin polynomial is strictly positive. The fact that $p$ is strictly positive and homogeneous in a given region can be used to accurately evaluate $p$ in a simple manner. We now present the proof from [46] that strictly positive homogeneous polynomials can be implemented with a finite bound on their relative error.
Accurate Evaluation of Strictly Positive Homogeneous Polynomials

Define the basic implementation as one which operates on the expanded form of the polynomial and first computes the monomials using a left to right logarithmic multiplication tree, then multiplies by each coefficient. Finally a left to right logarithmic addition tree is performed. For example a basic implementation of the Motzkin polynomial is:

\[
Basic(z^6 + x^2 y^2 (x^2 + y^2 - 3z^2)) = Basic(z^6 + x^4 y^2 + x^2 y^4 - 3x^2 y^2 z^2)
\]

\[
t_0 = ((z \hat{\times} z) \hat{\times} (z \hat{\times} z)) \hat{\times} (z \hat{\times} z)
\]

\[
t_1 = ((x \hat{\times} x) \hat{\times} (x \hat{\times} x)) \hat{\times} (y \hat{\times} y)
\]

\[
t_2 = ((x \hat{\times} x) \hat{\times} (y \hat{\times} y)) \hat{\times} (y \hat{\times} y)
\]

\[
t_3 = (-3) \hat{\times} (((x \hat{\times} x) \hat{\times} (y \hat{\times} y)) \hat{\times} (z \hat{\times} z))
\]

\[
Basic(z^6 + x^2 y^2 (x^2 + y^2 - 3z^2)) = (t_0 \hat{\times} t_1) \hat{\times} (t_2 \hat{\times} t_3)
\]

The error of this implementation can be bounded by the proof of Theorem 3.2 in [46]. Given that each floating-point operation introduces a relative error of \(1 + \delta\) for some \(\delta\) satisfying \(|\delta| < 2^{-mw}\) where \(mw\) is the output mantissa width of the operator, then this implementation returns a value which is:

\[
([z^6(1 + \delta_1)(1 + \delta_2)...(1 + \delta_5)]
\]

\[
x^4y^2(1 + \delta_6)(1 + \delta_7)...(1 + \delta_{10})(1 + \delta_{22})
\]

\[
+ [x^2y^4(1 + \delta_{11})(1 + \delta_{12})...(1 + \delta_{15})
\]

\[
- 3x^2y^2z^2(1 + \delta_{16})(1 + \delta_{17})...(1 + \delta_{21})(1 + \delta_{23})(1 + \delta_{24})
\]

\[
= z^6(1 + \delta_1)(1 + \delta_2)...(1 + \delta_5)(1 + \delta_{22})(1 + \delta_{24})
\]

\[
+ x^4y^2(1 + \delta_6)(1 + \delta_7)...(1 + \delta_{10})(1 + \delta_{22})(1 + \delta_{24})
\]

\[
+ x^2y^4(1 + \delta_{11})(1 + \delta_{12})...(1 + \delta_{15})(1 + \delta_{23})(1 + \delta_{24})
\]

\[
- 3x^2y^2z^2(1 + \delta_{16})(1 + \delta_{17})...(1 + \delta_{21})(1 + \delta_{23})(1 + \delta_{24})
\]

In general, if the polynomial is homogeneous of degree \(m\) and has \(s\) terms, then there will be \(m + \lceil \log_2 s \rceil\) factors of the form \(1 + \delta\) multiplying each
term in this final expression. For a general polynomial of the form:

\[ p = \sum_{\alpha} c_{\alpha} x^{\alpha} \]

\[ Basic(p) = \sum_{\alpha} c_{\alpha} x^{\alpha} \prod_{i=1}^{m+ [\log_2 s]} (1 + \delta_{\alpha,i}) \]

The relative error of \( Basic(p) \) over \( p \) is then:

\[ \frac{|Basic(p) - p|}{p} = \frac{\left| \sum_{\alpha} c_{\alpha} x^{\alpha} \left( \prod_{i=1}^{m+ [\log_2 s]} (1 + \delta_{\alpha,i}) - 1 \right) \right|}{\sum_{\alpha} c_{\alpha} x^{\alpha}} \]

Now because \( p \) is homogeneous, every term in the numerator and denominator has the same total degree in variables \( x_i \). So scaling every variable \( x_i \) by the same amount will leave the relative error unchanged. Given the assumption that the evaluation is not occurring near the variety then \( x \neq 0 \). So scaling by the inverse of \( ||x|| = \sqrt{\sum_i x_i^2} \) results in:

\[ \frac{|Basic(p) - p|}{p} = \frac{\left| \sum_{\alpha} c_{\alpha} \left( \frac{x}{||x||} \right)^{\alpha} \left( \prod_{i=1}^{m+ [\log_2 s]} (1 + \delta_{\alpha,i}) - 1 \right) \right|}{\sum_{\alpha} c_{\alpha} \left( \frac{x}{||x||} \right)^{\alpha}} \]

Now because \( |x_i| \leq ||x|| \) the relative error can be bounded as:

\[ \frac{|Basic(p) - p|}{p} \leq \frac{\sum_{\alpha} |c_{\alpha}| \left( (1 + \delta)^{m+ [\log_2 s]} - 1 \right)}{p_{\text{min}}} \]

where \( \delta \) is the largest relative error the floating-point components can introduce and \( p_{\text{min}} \) is the smallest value of \( p \) obtainable when \( ||x|| = 1 \) and in domain of interest \( D \):

\[ p_{\text{min}} = \min (p(x) : ||x|| = 1, x \in D) \]

Thus the following relationship guarantees an accurate evaluation with relative error \( \eta \) using the basic implementation with floating-point components.
with relative accuracy $\delta$ on a domain $D$:

$$
\frac{\sum_{\alpha} |c_{\alpha}| \left( (1 + \delta)^{m+\lfloor \log_2 s \rfloor} - 1 \right)}{\min (p(x) : \|x\| = 1, x \in D)} < \eta
$$

Provided $D$ does not include any part of the variety, $p_{\min}$ will be strictly positive and accurate evaluation can be achieved.

We can now use this proof in the case of evaluating the Motzkin polynomial far from the variety. Consider the following domain $D$ with parameter $e > 0$, which guarantees there that there are no points close to the line $x = y = z$:

$$
D(e) = \{(x, y, z) : \\
x, y, z \geq 0 \quad // \text{positive octant} \\
x \leq 2y, \quad y \leq 2x, \quad x \leq 2z, \quad y \leq 2z \\
// \text{in the neighbourhood of the line } x = y = z \\
(x - z)^2 \geq e(x^2 + y^2 + z^2) \quad \text{or} \quad (y - z)^2 \geq e(x^2 + y^2 + z^2) \\
// \text{bounded away from the line } x = y = z
$$

It can then be asked, what is the smallest value of $e$ such that F32 operations with a relative accuracy of $2^{-23}$ can be used with the basic implementation to guarantee a relative accuracy of $2^{-10}$. More precisely for a basic implementation of the Motzkin polynomial ($m = 6$, $s = 4$, $\sum_{\alpha} |c_{\alpha}| = 6$):

$$
\min_{e > 0} \left( \frac{6 \left( (1 + 2^{-23})^8 - 1 \right)}{\min (p(x) : \|x\| = 1, x, y, z \geq 0, x \leq 2y, y \leq 2x, x \leq 2z, y \leq 2z, (x - z)^2 \geq e(x^2 + y^2 + z^2) \quad \text{or} \quad (y - z)^2 \geq e(x^2 + y^2 + z^2))} < 2^{-10} \right)
$$

The minimum of $p$ will occur on the intersection of the unit sphere and the curves defined by $(x - z)^2 = e^2(x^2 + y^2 + z^2)$ and $(y - z)^2 = e^2(x^2 + y^2 + z^2)$. These curves are represented in Figure 7.7.

This minimisation is an exercise in polynomial minimisation which can
be performed via Gröbner bases and can be found in Appendix C.1. Such minimisations can be performed for all three lines forming the variety and for floating-point operators being F32 or F64 and can be found in Appendices C.2, D.1 and D.2. These constants can be found in Table 7.5.

There are regions that are not covered by these three subdomains, for example \(2x \leq y \leq 2z\). The only point on the variety that resides in these regions is the origin \(x = y = z = 0\). Close to the origin, but in regions not covered by the subdomains, a parameter \(e_5\) can be chosen to permit F32 operations to be used, provided one of the following hold:

\[
x^2 \geq e_5^2(x^2 + y^2 + z^2) \quad \text{or} \quad y^2 \geq e_5^2(x^2 + y^2 + z^2) \quad \text{or} \quad z^2 \geq e_6^2(x^2 + y^2 + z^2)
\]

Similarly \(e_6\) can be calculated to permit F64 operations. These calculations
Table 7.5.: Values of $e$ for Lines of the Variety and F32 and F64.

<table>
<thead>
<tr>
<th>Line</th>
<th>F32</th>
<th>F64</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = y = z$</td>
<td>$e_1 = 4860985 \times 2^{-25}$ $\approx 0.1448686421$</td>
<td>$e_2 = 12582933 \times 2^{-41}$ $\approx 0.000005722055448$</td>
</tr>
<tr>
<td>$z = x = 0$</td>
<td>$e_3 = 14247057 \times 2^{-25}$ $\approx 0.4245953858$</td>
<td>$e_4 = 14529599 \times 2^{-42}$ $\approx 0.000003303648327$</td>
</tr>
<tr>
<td>$z = y = 0$</td>
<td>$e_3 = 14247057 \times 2^{-25}$ $\approx 0.4245953858$</td>
<td>$e_4 = 14529599 \times 2^{-42}$ $\approx 0.000003303648327$</td>
</tr>
</tbody>
</table>

can be found in Appendices E.1 and E.2 and produce the following results:

$$e_5 = 2053059 \times 2^{-21} \approx 0.9789748192$$

$$e_6 = 15817739 \times 2^{-24} \approx 0.9428107142$$

7.4.3. Accurate Evaluation Close to the Variety

Consider using the basic evaluation scheme on the line of the variety $x = y = z$:

$$t_0 = ((x \times x) \times (x \times x)) \times (x \times x)$$
$$t_1 = ((x \times x) \times (x \times x)) \times (x \times x)$$
$$t_2 = ((x \times x) \times (x \times x)) \times (x \times x)$$
$$t_3 = (-3) \times (((x \times x) \times (x \times x)) \times (x \times x))$$

Basic($x^6 + x^2x^2(x^2 + x^2 - 3x^2)$) = ($t_0 \hat{+} t_1 \hat{+} t_2 \hat{+} t_3$)

Basic($x^6 + x^2x^2(x^2 + x^2 - 3x^2)$) = ($t_0 \hat{+} t_0 \hat{+} (t_0 \hat{+} (-3) \times t_0)$)

Basic($x^6 + x^2x^2(x^2 + x^2 - 3x^2)$) = (2$t_0 \hat{+} t_0 \hat{+} (t_0 \hat{+} (-3) \times t_0)$)

This must return zero if the implementation is to be relatively accurate. This requires:

$$(t_0 \hat{+} t_0) \hat{+} (t_0 \hat{+} (-3) \times t_0) \equiv 0$$
$$\Rightarrow (2t_0) \hat{+} (t_0 \hat{+} (-3) \times t_0) \equiv 0 \quad // \quad t_0 + t_0 \text{ is exactly representable}$$
$$\Rightarrow t_0 \hat{+} (-3) \times t_0 \equiv -2t_0$$
$$\Rightarrow (-3) \times t_0 \equiv -3t_0$$

This means that no rounding is permitted during the computation of $-3t_0$, but if representing $t_0$ requires all the bits of precision, then this is impossible.
The solution to this problem is to expand \( p \) about the particular line of the variety. In the case of the line \( x = y = z \) this can be achieved by rewriting \( p \) in terms of \( a = x - z \) and \( b = y - z \). Using the elimination techniques presented on page 70 in the preliminaries chapter, the result is:

\[
\begin{align*}
a &= x - z \\
\quad \quad b &= y - z \\
p(z, a, b) &= 4z^4a^2 + 4z^4ab + 4z^4b^2 \\
\quad \quad &+ 4z^3a^3 + 10z^3a^2b + 10z^3ab^2 + 4z^3b^3 \\
\quad \quad &+ z^2a^4 + 8z^2a^3b + 9z^2a^2b^2 + 8z^2ab^3 + z^2b^4 \\
\quad \quad &+ 2za^4b + 4za^3b^2 + 4za^2b^3 + 2zab^4 \\
\quad \quad &+ a^4b^2 + a^2b^4
\end{align*}
\]

Note that if the inputs are on the line \( x = y = z \) then \( a = b = 0 \) and this form of \( p \) evaluates to 0 as required. The behaviour of this form of \( p \) close to \( a = b = 0 \) is crucial in determining whether the polynomial can be accurately evaluated close to the variety. Consider the polynomial:

\[
q = (u^4 + v^4) + (v^2 + v^2)(x + y + z)^2
\]

Its variety is \( u = v = 0 \), however it cannot be accurately evaluated as close to \( u = v = 0 \) the term \( (v^2 + v^2)(x + y + z)^2 \) dominates over \( (u^4 + v^4) \) and \( (v^2 + v^2)(x + y + z)^2 \) cannot be accurately evaluated because it includes the unallowable variety \( x + y + z = 0 \). In order to determine the terms that dominate in \( p(z, a, b) \) as \( a, b \to 0 \) consider Figure 7.8 which contains all points \((i, j)\) where a term of the form \( a^ib^j \), possibly multiplied by a power of \( z \) exists in \( p(z, a, b) \).

The red outline indicates the convex hull of these points. The terms associated with the face of the convex hull nearest the origin dominate all other terms. For example, the term \( 4z^4ab \) will tend to zero slower than \( 9z^2a^2b^2 \) as \( a \) and \( b \) approach zero. Hence \( 4z^4ab \) dominates \( 9z^2a^2b^2 \). Similarly, the set of terms associated with the face nearest the origin dominate all others. The terms are:

\[
p_{\text{dom}} = 4z^4a^2 + 4z^4ab + 4z^4b^2
\]
The variety of this polynomial is $z = 0$ or $a = b = 0$, so this polynomial is accurately evaluable. Note however, that $p_{dom}$ has a larger variety than $p$ in that as well as $a = b = 0$ it also has $z = 0$. But note that the subdomain that contains the line $x = y = z$ does not include the $z$ axis so this extra variety will not affect the evaluation. If $p_{dom}$ is to be used as the evaluation of $p$ in the neighbourhood of $x = y = z$ and $z > 0$ then a bound on the relative error is required. The relative error can be bounded as follows:

$$
\left| \frac{Basic(p_{dom}) - p}{p} \right| \leq \left| \frac{Basic(p_{dom}) - p_{dom}}{p} \right| + \left| \frac{p_{dom} - p}{p} \right|
$$

Now if the following can be determined analytically on the domain of interest:

$$
\Delta = \max \left| \frac{p_{dom} - p}{p} \right|
$$

Then:

$$
\left| \frac{Basic(p_{dom}) - p}{p} \right| \leq \left| \frac{Basic(p_{dom}) - p_{dom}}{p_{dom}} \right| \left(1 + \frac{p_{dom} - p}{p}\right) + \Delta
$$

$$
\left| \frac{Basic(p_{dom}) - p}{p} \right| \leq \left| \frac{Basic(p_{dom}) - p_{dom}}{p_{dom}} \right| (1 + \Delta) + \Delta
$$

Figure 7.8.: Points $(i, j)$ if $a^i b^j$ exists in $p(z, a, b)$. 

\[\text{Figure 7.8.: Points } (i, j) \text{ if } a^i b^j \text{ exists in } p(z, a, b).\]
If this is to satisfy the global relative error requirement, then:

\[
\left| \frac{Basic(p_{dom}) - p}{p} \right| \leq \left| \frac{Basic(p_{dom}) - p_{dom}}{p_{dom}} \right| (1 + \Delta) + \Delta < \eta
\]

Rearranging:

\[
\left| \frac{Basic(p_{dom}) - p_{dom}}{p_{dom}} \right| < \frac{\eta - \Delta}{1 + \Delta}
\]

So if \( p_{dom} \) can be implemented with relative accuracy bounded above by \((\eta - \Delta)/(1 + \Delta)\) then \( Basic(p_{dom}) \) can be used as the accurate evaluation of \( p \) in this region. The calculation of \( \Delta \) can be found in Appendix F.1. The value of \( \Delta \) is:

\[
\Delta = \frac{6356645}{2^{39}} \approx 0.00001156266990
\]

Therefore it is required to implement \( p_{dom} \) with a relative accuracy of:

\[
\left| \frac{Basic(p_{dom}) - p_{dom}}{p_{dom}} \right| < \frac{\eta - \Delta}{1 + \Delta} \approx 0.0009649886723
\]

Recalling the condition for a basic implementation to be relatively accurate:

\[
\frac{\sum_{\alpha} |c_{\alpha}| \left( (1 + \delta)^{m + \lceil \log_2 s \rceil} - 1 \right)}{\min (p(x) : \|x\| = 1, x \in D)} < \eta
\]

Now this domain includes the variety, so \( p_{min} = 0 \), but on the variety \( a = b = 0 \) and \( Basic(p) = 0 \) as required. The calculation of \( p_{min} \) just off the variety can be found in Appendix F.2 and returns a minimum value of \( \frac{70368744177664}{199903912851459264513} \). So a basic implementation of \( p_{dom} \) must use components that satisfy:

\[
\frac{12 \left( (1 + \delta)^{10} - 1 \right)}{\frac{70368744177664}{199903912851459264513}} < \frac{\eta - \Delta}{1 + \Delta} \approx 0.0009649886723
\]

Note that the number of \( 1 + \delta \) terms includes the error in the computation of \( a \) and \( b \). The largest value of \( \delta \) which satisfies this is \( 2^{-42} \) so F64 operations will suffice.

Repeating the exercise for the other subdomains shows that near the \( y \)
axis \( p_{dom} = z^6 + x^2y^4 \), the associated \( \Delta = \frac{1512937}{2^{147}} \) and \( p_{min} = \frac{8413227}{2^{147}} \). Calculations can be found in Appendices G.1 and G.2. The precision of the operators must then satisfy:

\[
\frac{2 \left( (1 + \delta)^6 - 1 \right)}{\frac{8413227}{2^{147}}} < \eta - \Delta \frac{1 + \Delta}{1 + \Delta}
\]

The largest value of \( \delta \) that satisfies this is \( 2^{-139} \).

Finally in the regions not covered by the subdomains, as the origin is approached, all terms are dominant because \( p \) is homogeneous. The calculation of \( p_{min} \) for this region can be found in Appendix H and returns a value of \( p_{min} = \frac{236}{75} - \frac{32\sqrt{6}}{25} \approx 0.011319796 \). A basic implementation of \( p \) in this domain must then require using floating-point operations with a precision that satisfies:

\[
\frac{6 \left( (1 + \delta)^8 - 1 \right)}{\frac{236}{75} - \frac{32\sqrt{6}}{25}} < 2^{-10}
\]

The largest value of \( \delta \) which satisfies this is \( 2^{-23} \) so F32 operations will suffice.

### 7.4.4. Dealing with Floating-Point Error in the Branch Conditions

Having established the precision required for all regions, the algorithm which performs accurate evaluation of the Motzkin polynomial can nearly be presented. However, the conditions which define the various regions for which different precisions are required will also suffer from floating-point error. The conditions which describe the subdomains require the lossless floating-
point operations:

Subdomain for $x = y = z$

\[
x \leq 2y \quad y \leq 2x \quad x \leq 2z \quad y \leq 2z
\]

Subdomain for $z = x = 0$

\[
2x \leq y \quad 2z \leq y
\]

Subdomain for $z = y = 0$

\[
2y \leq x \quad 2z \leq x
\]

Note that these operations, namely multiplication by 2 and comparison operations, can be performed losslessly in floating-point arithmetic (provided that $2x$ is representable). Within each subdomain, further subdivisions are of the form:

\[
x^2 \geq e^2(x^2 + y^2 + z^2)
\]

\[
(x - z)^2 \geq e^2(x^2 + y^2 + z^2)
\]

Incorrectly returning false to the comparison will result in greater precision being used than necessary, although this will not affect accurate evaluation. Incorrectly returning true to the comparison has the potential for the algorithm to incorrectly use a lower precision than is acceptable. This situation has to be prevented. This will occur if the left hand side of the comparison rounds up and the left hand side rounds down. Assuming these operations are being performed in F32, then the squaring operations of the input are lossless as F32 has more than double the precision of F16 operations. Given that each F32 addition introduces a worse case relative error of $(1 \pm 2^{-23})$ the following is the worst case roundings of the comparisons:

\[
x^2 \geq e^2((x^2 + y^2)(1 - 2^{-23}) + (z^2 + 0)(1 - 2^{-23}))(1 - 2^{-23})
\]

\[
(x - z)^2(1 + 2^{-23})^2 \geq e^2((x^2 + y^2)(1 - 2^{-23}) + (z^2 + 0)(1 - 2^{-23}))(1 - 2^{-23})
\]

Rearranging:

\[
x^2 \geq e^2(1 - 2^{-23})^2(x^2 + y^2 + z^2)
\]

\[
(x - z)^2 \geq e^2 \left( \frac{1 - 2^{-23}}{1 + 2^{-23}} \right)^2 (x^2 + y^2 + z^2)
\]

272
So if new error terms $e'$ are introduced which satisfy:

\[
(e')^2 = \left( \frac{e}{1 - 2^{-23}} \right)^2
\]
\[
(e')^2 = \left( \frac{e + 2^{-23}}{1 - 2^{-23}} \right)^2
\]

Respectively for the two cases, then no false positive can occur (note these values should then be rounded up to the nearest F32 value).

So the constants $e_1, e_2, ..., e_6$:

\[
e_1 = 4860985 \times 2^{-25}
\]
\[
e_2 = 12582933 \times 2^{-41}
\]
\[
e_3 = 14247057 \times 2^{-25}
\]
\[
e_4 = 14529599 \times 2^{-42}
\]
\[
e_5 = 2053059 \times 2^{-21}
\]
\[
e_6 = 15817739 \times 2^{-24}
\]

Need modifying as follows (these are rounded up to the nearest F32 value):

\[
(e_1')^2 = \left( \frac{e_1 + 2^{-23}}{1 - 2^{-23}} \right)^2
\]
\[
(e_2')^2 = \left( \frac{e_2 + 2^{-23}}{1 - 2^{-23}} \right)^2
\]
\[
(e_3')^2 = \left( \frac{e_3}{1 - 2^{-23}} \right)^2
\]
\[
(e_4')^2 = \left( \frac{e_4}{1 - 2^{-23}} \right)^2
\]
\[
(e_5')^2 = \left( \frac{e_5}{1 - 2^{-23}} \right)^2
\]
\[
(e_6')^2 = \left( \frac{e_6}{1 - 2^{-23}} \right)^2
\]
This results in:

\[
\begin{align*}
(e_1')^2 &= \frac{11267275}{2^{29}} \\
(e_2')^2 &= \frac{9437221}{2^{58}} \\
(e_3')^2 &= \frac{12098473}{2^{26}} \\
(e_4')^2 &= \frac{12583095}{2^{60}} \\
(e_5')^2 &= \frac{16079149}{2^{24}} \\
(e_6')^2 &= \frac{7456569}{2^{23}}
\end{align*}
\]

7.4.5. Algorithm for the Accurate Evaluation of the Motzkin Polynomial

Combining all the previous subsections together, an algorithm which accurately evaluates the Motzkin polynomial can now be presented. The implementation has the following features: conversions between floating-point formats using the RTE rounding mode. The function \( \text{Basic}N(p) \) will be used which implements a basic implementation using floating-point operators of size \( N \in \{32, 64\} \), e.g.

\[
\text{Basic}N(3x^2 + 2x^2y + 3xy^2) = (3((xx)x) + 2((xx)y)) + 3((xy)y)
\]

However, parts of the algorithm requires a function denoted \( \text{Basic}128^+(p) \), which is extended quad precision with a mantissa of length 139.
Inputs

$F_{16} \ x, y, z; \ // \ non\ exceptional\ and\ non\ denormal$

Output

$F_{32} \  \hat{p}; \ //\ will\ have\ a\ relative\ error\ of\ strictly\ less\ than\ 2^{-10}$

$F_{32} \ x = \lfloor x \rfloor;$
$F_{32} \ y = \lfloor y \rfloor;$
$F_{32} \ z = \lfloor z \rfloor;$
$F_{32} \ x^2 = x^2;$
$F_{32} \ y^2 = y^2;$
$F_{32} \ z^2 = z^2;$
$F_{32} \ a = x - z;$
$F_{32} \ b = y - z;$
$F_{32} \ c = x - y;$
$F_{32} \ a^2 = a^2;$
$F_{32} \ b^2 = b^2;$
$F_{32} \ s = (x^2 + y^2) + z^2;$
$F_{32} \ e_1 = \frac{11267275}{2^{20}};$
$F_{32} \ e_2 = \frac{9437221}{2^{58}};$
$F_{32} \ e_3 = \frac{12098473}{2^{26}};$
$F_{32} \ e_4 = \frac{12583095}{2^{60}};$
$F_{32} \ e_5 = \frac{16079149}{2^{24}};$
$F_{32} \ e_6 = \frac{7456569}{2^{21}};$
if \((x \leq 2y, y \leq 2x, x \leq 2z, y \leq 2z)\) \{ // near the line \(x = y = z\)

if \((a2 \geq e_1 s \ || \ b2 \geq e_1 s)\) // far enough from the variety for F32
\[
\hat{p} = \text{Basic32}(z^6 + x^4 y^2 + x^2 y^4 - 3x^2 y^2 z^2);
\]
elsif \((a2 \geq e_2 s \ || \ b2 \geq e_2 s)\) // far enough from the variety for F64
\[
\hat{p} = \text{Basic64}(z^6 + x^4 y^2 + x^2 y^4 - 3x^2 y^2 z^2);
\]
else // close to the subvariety
\[
\hat{p} = \text{Basic64}(4z^4 a^2 + 4z^4 ab + 4z^4 b^2);
\]
\}

elsif \((2x \leq y, 2z \leq y)\) \{ // near the line \(x = z = 0\)

if \((x2 \geq e_3 s \ || \ z2 \geq e_3 s)\) // far enough from the variety for F32
\[
\hat{p} = \text{Basic32}(z^6 + x^4 y^2 + x^2 y^4 - 3x^2 y^2 z^2);
\]
elsif \((x2 \geq e_4 s \ || \ z2 \geq e_4 s)\) // far enough from the variety for F64
\[
\hat{p} = \text{Basic64}(z^6 + x^4 y^2 + x^2 y^4 - 3x^2 y^2 z^2);
\]
else // close to the variety
\[
\hat{p} = \text{Basic128}^+(z^6 + x^2 y^4);
\]
\}

elsif \((2y \leq x, 2z \leq x)\) \{ // near the line \(y = z = 0\)

if \((y2 \geq e_3 s \ || \ z2 \geq e_3 s)\) // far enough from the variety for F32
\[
\hat{p} = \text{Basic32}(z^6 + x^4 y^2 + x^2 y^4 - 3x^2 y^2 z^2);
\]
elsif \((y2 \geq e_4 s \ || \ z2 \geq e_4 s)\) // far enough from the variety for F64
\[
\hat{p} = \text{Basic64}(z^6 + x^4 y^2 + x^2 y^4 - 3x^2 y^2 z^2);
\]
else // close to the variety
\[
\hat{p} = \text{Basic128}^+(z^6 + y^2 x^4);
\]
\}

else \{ // near no part of the variety

if \((x2 \geq e_5 s \ || \ y2 \geq e_5 s \ || \ z2 \geq e_5 s)\) // far enough for F32
\[
\hat{p} = \text{Basic32}(z^6 + x^4 y^2 + x^2 y^4 - 3x^2 y^2 z^2);
\]
elsif \((x2 \geq e_6 s \ || \ y2 \geq e_6 s \ || \ z2 \geq e_6 s)\) // far enough for F64
\[
\hat{p} = \text{Basic64}(z^6 + x^4 y^2 + x^2 y^4 - 3x^2 y^2 z^2);
\]
else
\[
\hat{p} = \text{Basic32}(z^6 + x^4 y^2 + x^2 y^4 - 3x^2 y^2 z^2);
\]
\}
This algorithm has been verified by extensive simulation with 1 billion sets of random input stimuli, with directed tests hitting each of the branches and stressing the inputs which lie on the borders between the various branches. Given the floating-point components are faithfully rounded, each operation can return two possible answers (unless the answer is representable in the target floating-point format). During each simulation, both possible answers are calculated at each operation and propagated through the rest of the algorithm. The infinitely precise model used is that of an interval arithmetic based MPFR [60] model with customisable precision. The inferred likelihood of entering each of the branches can be found in an annotated version of the algorithm on the next page. This demonstrates that F32 operations can be used for 37% of inputs, 59% for F64 operations and 4% for F128+ operations. The detailed error analysis has enabled us to create an algorithm that attempts to use just enough precision for each subdomain.

As an example implementation of this algorithm, the components required for its design, including three faithfully rounded floating-point adders and three faithfully rounded floating-point multipliers were constructed with precisions F32, F64 and F128+ respectively. A sample logic synthesis of this collection of hardware components resulted in an area of 189\(\mu\text{m}^2\) for six pipeline stages with a clock period of 2.5ns in the TSMC 40nm standard cell library Tcbn40lpbwp. A total latency of 21 clock cycles would be required to produce an accurate evaluation of the Motzkin polynomial using this hardware. Crucially, the bulk of the area, 85%, is for the F128+ operations, which will only be used in 4% of cases. This is the only known hardware implementation that can produce such an accurate evaluation.
if \( x \leq 2y, y \leq 2x, x \leq 2z, y \leq 2z \) \{ // near the line \( x = y = z \)
  if \( a2 \geq c1s \ \| \ b2 \geq e1s \) // far enough from the variety for F32
    \( \hat{p} = \text{Basic32}(z^6 + x^4y^2 + x^2y^4 - 3x^2y^2z^2) \); 3.3%
  elsif \( a2 \geq c2s \ \| \ b2 \geq e2s \) // far enough from the variety for F64
    \( \hat{p} = \text{Basic64}(z^6 + x^4y^2 + x^2y^4 - 3x^2y^2z^2) \); 0.1%
  else // close to the subvariety
    \( \hat{p} = \text{Basic64}(4z^4a^2 + 4z^4ab + 4z^4b^2) \); 3.3%
  \}
elsif \( 2x \leq y, 2z \leq y \) \{ // near the line \( x = z = 0 \)
  if \( x2 \geq c3s \ \| \ z2 \geq e3s \) // far enough from the variety for F32
    \( \hat{p} = \text{Basic32}(z^6 + x^4y^2 + x^2y^4 - 3x^2y^2z^2) \); 0.3%
  elsif \( x2 \geq e4s \ \| \ z2 \geq e4s \) // far enough from the variety for F64
    \( \hat{p} = \text{Basic64}(z^6 + x^4y^2 + x^2y^4 - 3x^2y^2z^2) \); 26.9%
  else // close to the variety
    \( \hat{p} = \text{Basic128}^+(z^6 + x^2y^4) \); 2.0%
  \}
elsif \( 2y \leq x, 2z \leq x \) \{ // near the line \( y = z = 0 \)
  if \( y2 \geq c3s \ \| \ z2 \geq e3s \) // far enough from the variety for F32
    \( \hat{p} = \text{Basic32}(z^6 + x^4y^2 + x^2y^4 - 3x^2y^2z^2) \); 0.3%
  elsif \( y2 \geq e4s \ \| \ z2 \geq e4s \) // far enough from the variety for F64
    \( \hat{p} = \text{Basic64}(z^6 + x^4y^2 + x^2y^4 - 3x^2y^2z^2) \); 26.9%
  else // close to the variety
    \( \hat{p} = \text{Basic128}^+(z^6 + y^2x^4) \); 2.0%
  \}
else \{ // near no part of the variety
  if \( x2 \geq e5s \ \| \ y2 \geq e5s \ \| \ z2 \geq e5s \) // far enough for F32
    \( \hat{p} = \text{Basic32}(z^6 + x^4y^2 + x^2y^4 - 3x^2y^2z^2) \); 22.8%
  elsif \( x2 \geq e6s \ \| \ y2 \geq e6s \ \| \ z2 \geq e6s \) // far enough for F64
    \( \hat{p} = \text{Basic64}(z^6 + x^4y^2 + x^2y^4 - 3x^2y^2z^2) \); 2.0%
  else
    \( \hat{p} = \text{Basic32}(z^6 + x^4y^2 + x^2y^4 - 3x^2y^2z^2) \); 10.3%
  \}
7.5. Towards an Algorithm in the General Case

In the pursuit of an algorithm which produces an accurate evaluation of a
general polynomial, the previous section presented a proof of concept by pro-
viding a complete worked example of the Motzkin polynomial. The Motzkin
polynomial is irreducible and has a non trivial variety, thus the implementa-
tion in the previous section is a significant demonstration of the practicality
of the framework put forward in [46]. For a general polynomial the steps
required in the creation of an accurately evaluating implementation include
the calculation of the variety, the creation of the subdomains, determin-
ing the $e_i$ parameters and the dominant terms and precisions required for
accurate evaluation close to the subvarieties. This section discusses these
challenges and contributes to their solution.

The approach taken in [46] requires the polynomial to be irreducible and
homogeneous. These limitations can be easily overcome by first factorising
a polynomial into irreducible polynomials, which can be performed by com-
puter algebra packages such as Singular [43]. Secondly, one can homogenise
an arbitrary polynomial $p$ by evaluating polynomial $q$ which has an extra
variable $x_0$ setting $x_0 = 1$ where ($d$ is the largest total degree of $p$):

$$q(x_0, x_1, x_2, ..., x_n) = x_0^d p \left( \frac{x_1}{x_0}, \frac{x_2}{x_0}, ..., \frac{x_n}{x_0} \right)$$

Note that $q$ is in fact homogeneous, consider:

$$q(\lambda x_0, \lambda x_1, \lambda x_2, ..., \lambda x_n) = (\lambda x_0)^d p \left( \frac{\lambda x_1}{\lambda x_0}, \frac{\lambda x_2}{\lambda x_0}, ..., \frac{\lambda x_n}{\lambda x_0} \right)$$

$$= \lambda^d x_0^d p \left( \frac{x_1}{x_0}, \frac{x_2}{x_0}, ..., \frac{x_n}{x_0} \right)$$

$$= \lambda^d q$$

Hence $q$ is homogeneous. The generalisation of each step used in the Motzkin
example is now addressed, these include determining the variety of the poly-
nomial, input domain partitioning, accurate evaluation far from the vari-
eties, accurate evaluation close to the variety and dealing with floating-point
errors in the branch conditions.
7.5.1. Determining the Allowable Nature of the Variety, 
Automatic Test Pattern Generation and Component 
Suggestion

The variety of the Motzkin polynomial was established by an application of 
the arithmetic-geometric mean property. In general, determining the real 
variety of a polynomial can be achieved via the use of a computer algebra 
package such as Maple [122] and the command \textit{SemiAlgebraic} which returns 
the conditions on the real inputs such that a given polynomial is zero. If the 
variety is allowable then the result must reduce to the union of intersections 
of conditions of the form \( x_i = 0 \) or \( x_j \pm x_k = 0 \) for some inputs \( x_i, x_j \) 
and \( x_k \). If the variety is unallowable then some of the conditions returned 
will be not of this form, say \( f(x) = 0 \). Now \( f \) can be returned to the user 
with a statement that without creating a floating-point component which 
computes the entirety of \( f \) with bounded relative error, accurate evaluation 
of \( p \) is impossible. Further, inputs which satisfy \( f = 0 \) are likely to stress 
any floating-point implementation of \( p \). This fact can be used for automatic 
test pattern generation. Choose any of the inputs to \( f \), say \( x_i \), choose 
any floating-point values for the other inputs, solve \( f(x) = 0 \) then find 
the nearest representable floating-point numbers to this infinitely precise 
\( x_i \). Such inputs will lie just off the variety and are likely to be difficult to 
accurately evaluate by any floating-point implementation.

This general technique works in the case of the Motzkin polynomial as 
Maple returns the following variety decomposition:

\[
\begin{align*}
&> \text{with(SolveTools)} \\
&> \text{simplify(SemiAlgebraic({z^6+x^2*y^2*(x^2+y^2-3*z^2) = 0}))} \\
&\text{assuming z::real}
\end{align*}
\]

\[
[ [z < 0, y = z, x = z], \\
[z < 0, y = z, x = -z], \\
[z < 0, y = -z, x = z], \\
[z < 0, y = -z, x = -z], \\
[z = 0, y < 0, x = 0], \\
[z = 0, y = 0, x = x], \\
[z = 0, 0 < y, x = 0], \\
[0 < z, y = -z, x = -z], \\
[0 < z, y = -z, x = z],
\]

280
These conditions match the previously calculated variety $z = x = 0$, $z = y = 0$ and $\pm x = \pm y = \pm z$. In the case of a polynomial with an unallowable variety, say the following polynomial:

$$q = ba(b^2 - a^2) + cb(c^2 - b^2) + ac(a^2 - c^2)$$

One of the conditions returned by Maple is:

$$[c < 0, b < c, a = -c-b]$$

Hence the user can be informed that without a three input floating-point adder accurate evaluation of $q$ is impossible. Further, choosing any floating-point values for $b$ and $c$, then find the infinitely precise number $a$ such that $a + b + c = 0$, then finding the nearest representable floating-point numbers to this $a$ value will create a set of input values that would stress any floating-point implementation of $q$.

### 7.5.2. Input Domain Partitioning

If the polynomial has an allowable variety, then the input domain has to be partitioned as in Section 7.4.1. This process can be placed on an algorithmic footing. An allowable variety is the union of parts of the following form:

$$x_{i_1} = x_{i_2} = ... = x_{i_r} = 0 \quad // \text{for some indices } i$$

$$(-1)^{s_{j_1}} x_{j_1} = (-1)^{s_{j_2}} x_{j_2} = ... = (-1)^{s_{j_m}} x_{j_m} \quad // \text{for some indices } j \text{ and constants } s$$

Each of these parts is called a subvariety. The Motzkin example proceeded by projecting each of the parts of the variety onto each possible plane formed by each possible pair of input variables. Then on each plane constraints are created that separate a given subvariety from all the others. The complete set of possible conditions which separate part of variety of interest A from another B when projected onto the $x, y$ plane can be found in Table 7.6.
Table 7.6: Constraints for Input Domain Partitioning.

<table>
<thead>
<tr>
<th>Part of the Variety A</th>
<th>Part of the Variety B</th>
<th>Resulting Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = \pm y)</td>
<td>(x = \pm y)</td>
<td>none</td>
</tr>
<tr>
<td>(x = y)</td>
<td>(x = -y)</td>
<td>(\text{sgn}(x) = \text{sgn}(y))</td>
</tr>
<tr>
<td>(x = -y)</td>
<td>(x = y)</td>
<td>(\text{sgn}(x)! = \text{sgn}(y))</td>
</tr>
<tr>
<td>(x = \pm y)</td>
<td>(x = 0)</td>
<td>(</td>
</tr>
<tr>
<td>(x = \pm y)</td>
<td>(y = 0)</td>
<td>(</td>
</tr>
<tr>
<td>(x = 0)</td>
<td>(x = \pm y)</td>
<td>(</td>
</tr>
<tr>
<td>(x = 0)</td>
<td>(x = 0)</td>
<td>none</td>
</tr>
<tr>
<td>(x = 0)</td>
<td>(y = 0)</td>
<td>(</td>
</tr>
<tr>
<td>(y = 0)</td>
<td>(x = \pm y)</td>
<td>(</td>
</tr>
<tr>
<td>(y = 0)</td>
<td>(x = 0)</td>
<td>(</td>
</tr>
<tr>
<td>(y = 0)</td>
<td>(y = 0)</td>
<td>none</td>
</tr>
<tr>
<td>any</td>
<td>(x = y = 0)</td>
<td>none</td>
</tr>
<tr>
<td>(x = y = 0)</td>
<td>any</td>
<td>none</td>
</tr>
</tbody>
</table>

Say the variety is the union of \(k\) subvarieties, then the algorithm which creates the definition of the subdomain which separates the \(i\)th subvariety from the others, is as follows (note that all of these constraints can be computed losslessly using floating-point components):
Inputs

\( E_i = \) set of variables that are equal, up to their sign, in the \( i \)th subvariety

\( S(i, x) = \) returns the sign of variable \( x \) in the set of variables which are equal in the \( i \)th part of the variety

\( Z_i = \) set of variables which are zero in the \( i \)th part of the variety

Outputs Constrs — constraints defining the subdomain around the \( i \)th subvariety

\( Constrs = \{ \} \)

For all possible pairs \( x, y \in \{x_1, x_2, ..., x_n\} \) \{ 

\begin{align*}
\text{if } & \quad ((x \in E_i) \land (y \in E_i) \land ((x \notin Z_i) \lor (y \notin Z_i))) \quad \{ \\
\text{for } & \quad j = 1..k \quad j \neq i \quad \{ \\
& \quad \text{if } (x, y \in E_i \land y \in E_j \land S(i, x) = S(i, y) \land S(i, x) \neq S(i, y)) \\
& \quad \quad Constrs \quad \cup = \quad \{ \text{sgn}(x) = \text{sgn}(y) \} \\
& \quad \text{if } (x, y \in E_i \land x \in Z_j) \\
& \quad \quad Constrs \quad \cup = \quad \{|y| \leq |2x|\} \\
& \quad \text{if } (x, y \in E_i \land y \in Z_j) \\
& \quad \quad Constrs \quad \cup = \quad \{|x| \leq |2y|\} \\
& \quad \text{if } (x \in Z_i \land y \in E_j) \\
& \quad \quad Constrs \quad \cup = \quad \{|2x| \leq |y|\} \\
& \quad \text{if } (y \in Z_i \land x \in E_j) \\
& \quad \quad Constrs \quad \cup = \quad \{|2y| \leq |x|\} \\
& \quad \text{if } (x \in Z_i \land y \in Z_j) \\
& \quad \quad Constrs \quad \cup = \quad \{|x| \leq |y|\} \\
& \quad \text{if } (y \in Z_i \land z \in Z_j) \\
& \quad \quad Constrs \quad \cup = \quad \{|y| \leq |x|\} \\
\} \}
\end{align*}
7.5.3. General Accurate Evaluation Far from the Variety

Recall from equation (7.1) on page 265 that accurate evaluation of the polynomial \( p = \sum_{\alpha} c_{\alpha} x^{\alpha} \) with relative error bounded by \( \eta \) far from the variety can be achieved using the basic implementation of the polynomial which is homogeneous of total degree \( m \) with \( s \) terms using floating-point components with bounded relative error \( \delta \) on a domain \( D \) provided:

\[
\frac{\sum_{\alpha} |c_{\alpha}| \left((1 + \delta)^{m + \lceil \log_2 s \rceil} - 1\right)}{\min(p(x) : \|x\| = 1, x \in D)} < \eta
\]

(7.2)

where the domain \( D \) is in one of the subdomains and bounded away from the variety. From the previous section a definition of a subdomain can be written as the intersection of constraints of the form:

\[
sgn(x_i) == sgn(x_j) \quad sgn(x_i)! = sgn(x_j) \quad |x_i| \leq |x_j| \quad |x_i| \leq 2|x_j| \quad 2|x_i| \leq |x_j|
\]

By splitting across all the orthants, these conditions can be decomposed into a union of intersections of constraints of the form:

\[
x_i \leq 0 \quad 0 \leq x_j \quad x_i \leq \pm x_j \quad x_i \leq \pm 2x_j \quad 2x_i \leq \pm x_j
\]

The conditions for being bounded away from the variety of unions of constraints of the form:

\[
x_i^2 \geq e^2(x_1^2 + x_2^2 + \ldots + x_n^2) \quad (x_i - x_j)^2 \geq e^2(x_1^2 + x_2^2 + \ldots + x_n^2)
\]

Due to the constraint of being on the unit sphere, these reduce to a union of conditions of the form:

\[
\pm x_i \geq e \quad (x_i - x_j) \geq e
\]

Therefore \( D \) is the union of intersections of constraints which include inequalities on \( x_i \) and a constraint of the form \( \pm x_i \geq e \) or \( (x_i - x_j) \geq e \). Now equation (7.2) can be rearranged as follows:

\[
\frac{\sum_{\alpha} |c_{\alpha}| \left((1 + \delta)^{m + \lceil \log_2 s \rceil} - 1\right)}{\eta} < \min(p(x) : \|x\| = 1, x \in D)
\]
(Note that Lemma 4.45 in [46] proves that non linear irreducible homogeneous polynomials with allowable varieties do not change their sign so without loss of generality $p$ can be assumed to be non negative as $-p$ can be implemented and the sign bit of the floating-point output set to one. Linear irreducible polynomials with an allowable variety are polynomials which are either $x_i$ or $x_i \pm x_j$ which are trivially implementable with bounded relative error using the floating-point adder in the latter cases. Hence this rearrangement of equation (7.2) is valid.) This condition is met if the following conditions hold:

$$\sum_{\alpha} |c_{\alpha}| \left( \frac{(1 + \delta)^{m + \lfloor \log_2 s \rfloor} - 1}{\eta} \right) < p$$

$$x_1^2 + x_2^2 + ... + x_n^2 = 1$$

If these conditions are met then using a basic implementation of $p$ using floating-point components with a bounded relative error of $\delta$ will achieve an overall relative error bounded by $\eta$. For a given value of $\delta$ it is desired to know the smallest value of $e$ for which these conditions hold as closer to the variety a higher degree of precision is likely to be required. The optimisation problem is then:

$$\min e$$

subject to $$\sum_{\alpha} |c_{\alpha}| \left( \frac{(1 + \delta)^{m + \lfloor \log_2 s \rfloor} - 1}{\eta} \right) < p$$

$$x_1^2 + x_2^2 + ... + x_n^2 = 1$$

$$x \in D$$

285
Now $D$ is a union of intersections of constraints, let $G_i(x, e)$ be the $i$th set of intersecting constraints in the union. Now let:

$$e_i = \min e$$

subject to

$$\sum_{\alpha} |c_\alpha| \left( (1 + \delta)^{m + [\log_2 s]} - 1 \right) < p$$

$$x_1^2 + x_2^2 + ... + x_n^2 = 1$$

$$G_i(x, e)$$

The optimal value of $e$ is then the maximum of the $e_i$ variables. Crucially the optimisation problem calculating variable $e_i$ is in a form where the Karush–Kuhn–Tucker conditions [102] can be applied which will result in solution of a set of multivariate polynomial equations which can be solved via Gröbner bases techniques. Thus, in the general case, the condition for use of particular precision over a particular domain parameterised by such an $e$ variable can be calculated automatically.

7.5.4. General Accurate Evaluation Close to the Variety

Close to the variety the polynomial in question must be expanded using the variables defining the variety within a given subdomain, this can be achieved using the elimination techniques presented on page 70. Determining the set of dominant terms in general, requires the computation of the terms associated with the faces of the convex hull of a multi dimensional Newton polytope. In general there may be multiple dominant terms depending on the direction the variety is approached, a detailed error analysis of the branching conditions required to separate these directions is the subject of future work.

7.5.5. General Handling of Floating-Point Errors in the Branch Conditions

The conditions defining the subdomains require operations which include sign comparisons, floating-point comparisons and multiplication by two. All of these operations can be performed without any errors being introduced. The conditions bounding evaluation away from the variety will be of one of
the following two forms:

\[ x_i^2 \geq e^2(x_1^2 + x_2^2 + \ldots + x_n^2) \quad (x_i - x_j)^2 \geq e^2(x_1^2 + x_2^2 + \ldots + x_n^2) \]

Following the same line of reasoning put forward in Section 7.4.4, the use of floating-point components with bounded relative error of \( \delta \) will only guarantee the following bounds:

\[ x_i^2 \geq e^2(1 - \delta)^{\lceil \log_2 n \rceil} (x_1^2 + x_2^2 + \ldots + x_n^2) \]
\[ (x_i - x_j)^2 \geq e^2(1 - \delta)^{\lceil \log_2 n \rceil} \frac{1}{(1 + \delta)^2} (x_1^2 + x_2^2 + \ldots + x_n^2) \]

By replacing \( e \) by the following values, conservatively rounded up to precision \( \delta \), will ensure the original bounds are guaranteed to be met regardless of the floating-point errors introduced:

\[ e' = \frac{e}{(1 - \delta)^{\lceil \log_2 n \rceil}/2} \]
\[ e' = \frac{e(1 + \delta)}{(1 - \delta)^{\lceil \log_2 n \rceil}/2} \]

7.6. Conclusion

This chapter has addressed the creation of lossy floating-point polynomial implementations. We have put forward a construction of floating-point multipliers which have bounded relative error and are up to 54% smaller than correctly rounded multipliers by leveraging the work on the creation of faithfully rounded integer multipliers from Chapter 5. Lossy synthesis requires the control of errors introduced, this lead us to investigate the work of Demmel et al. [46] which provided a framework for calculating polynomials with bounded relative error. The framework was non constructive in nature, this chapter has provided the algorithmic methods to demonstrate that the approach can work in practice by fully completing a working implementation of the Motzkin polynomial. This approach has the potential to create implementations with guaranteed bounded relative error. The barrier to full automation concerns the general case of accurate evaluation close to the variety and is the subject of future work.
8. Conclusion

This thesis has addressed problems in polynomial ASIC datapath design, with a focus on a lossy design paradigm where a user defined acceptable error is exploited to minimise hardware implementation costs. A crucial feature of a lossy flow is that the errors introduced are controlled such that a user defined error specification is guaranteed to be satisfied by the results of the lossy procedure. Approaches that do not offer guarantees on any errors introduced are of limited value to designers as this forces a difficult verification problem on the user; such approaches will have very limited practical use. For this reason a highly analytic approach has been taken throughout this thesis. Control of the error is essential, second to this is the desire to fully exploit the error freedom. Where possible, attempts have been made to pose and solve optimisation problems which result in maximal error exploitation. Formal verification techniques pervade hardware design, every tool usage invariably has an associated methodology to independently establish the correctness of the output of the tool, for example logical equivalence checking of RTL against gate level netlists. In light of this, the thesis has also introduced techniques that are capable of independently establishing the correctness of the transformations introduced. Where the design flows have the potential of creating multiple potential implementations, attempts have been made to anticipate the hardware quality of each, which can be used to guide the user selection.

Chapter 2 gave background on logic synthesis and an introduction to common implementations for integer addition and multiplication. The motivation and history for high level synthesis (HLS), word-length optimisation and existing imprecise or lossy operators were discussed. Given the ubiquity of polynomials and their manipulation throughout the thesis, Chapter 3 contained key results from algebraic geometry and established certain techniques and notation. Chapter 4 contained the first technical contribution of the thesis and addressed a lossless polynomial implementation problem.
Integer arithmetic logic units and floating-point operators contain datapath components which must implement mutually exclusive polynomials with integer coefficients and it is common for designers to optimise these designs by hand. Chapter 4 sought to establish a method for algorithmically optimising such designs. It was observed that an integer sum-of-product operation has a very efficient ASIC implementation and this fact was used to guide a method for optimising the mutually exclusive set. The technique presented included implementing only one polynomial, the control logic was minimised via the use of an integer program and a novel approach to optional negation was presented. The overall flow has the potential to be entirely automated, implementing the flow manually on a set of examples demonstrated that the flow can achieve a reduction in hardware implementation area up to 50%. In order to formally prove the correctness of the flow a method which involved the super usage of existing industry equivalence checking tools was introduced which was capable of making feasible currently infeasible verifications. Thus Chapter 4 provided a flow which fits into existing HLS flows with an associated formal verification method exhibiting considerable hardware implementation benefits. Chapter 4 can be seen as a pre-processing step, combining the required implementation of many polynomials into the implementation of one. Subsequent chapters focused on the implementation of that single polynomial.

Chapter 5 began to introduce the notion of lossy synthesis by showing how to build lossy fixed-point components. These building blocks are pieces of RTL which can be used off-the-shelf, exhibit lower hardware implementation costs than their correctly rounded counterparts and have guaranteed worst case absolute error by guaranteeing to be a faithful rounding. Three types of components were presented: integer multipliers, multioperand adders (arbitrary binary array) and constant divisions. These components are a significant contribution of the thesis, existing methods for constructing such components with reduced accuracy typically gave no guarantees on their error properties. Further, these components were all shown to be optimal in the sense that they are the best architecture of a given type which fully exploit the error freedom. The ability to produce a lossy version of an arbitrary binary array gave rise to new truncated multipliers. In order to create these components with guarantees on their error, a high degree of analytic effort had to be expended. The resultant conditions, however, were simple
enough to embed directly into RTL. The resultant components exhibited up to 46% smaller hardware area for multiplication and 50% for constant division. In addition to their creation, a method was presented which could independently formally verify the components using industry tools. Thus Chapter 5 offers designers ready-to-use components and a method for independently establishing their correctness.

Chapter 6 then addresses the question of implementing a fixed-point polynomial with rational coefficients. Any such polynomial can be constructed using sum-of-product operations and constant divisions and the techniques and components from Chapter 5 allow for the exploitation of an absolute error for hardware implementation cost reduction of such components. Chapter 6 begins by generalising the faithfully rounded components of Chapter 5 to cope with an arbitrary absolute maximum error. It is then assumed that a given configuration of sum-of-product and constant division operations has been chosen. A heuristic is then presented which attempts to establish how best to allocate a maximum absolute error at each node of the configuration while maintaining a global user defined absolute error bound. The heuristic has the potential to produce a large number of design implementation candidates and it would be useful to determine their hardware implementation properties without having to perform logic synthesis on all of them. To this end, a hardware area cost heuristic is developed which seeks to correctly ascertain which of the competing implementations will have smallest hardware area. Experimentation demonstrated up to 47% area reduction of the presented technique over word length optimisation and the area cost heuristic created was shown to be able to successfully determine the smallest design within sets of competing designs with high confidence. Thus Chapter 6 provides a method for producing candidate implementations of fixed-point polynomials with rational coefficients with a guaranteed maximum absolute error bound and a heuristic area cost function to guide selection.

The final technical chapter, Chapter 7, moved the attention from fixed-point to floating-point polynomials. Controlling the accuracy of floating-point implementations is notoriously difficult. The goal of lossy synthesis is to introduce error in a controlled manner and, given the floating-point format, it is natural is ask whether it is possible to create an algorithm which implements a floating-point polynomial with guaranteed relative error as opposed to the absolute error goal used in the fixed-point chapters. There
Chapter 7 introduced the floating-point format and the variety of methods used to evaluate polynomials and then showed how the work from Chapter 5 on faithfully rounded integer multipliers can be used to create faithfully rounded floating-point multipliers which are up to 54% smaller than their correctly rounded counterparts. Again this component can be used as an off-the-shelf RTL component with guaranteed worst case relative error and is also a significant contribution of this thesis. (An appendix showed how a faithfully rounded floating-point adder can be constructed which is up to 73% smaller the equivalent correctly rounded component.) Having constructed these components the only approach for accurate evaluation is the framework proposed in [46]. However, this lacked a constructive approach and a complete error analysis, Chapter 7 completed this work and provided an example in the form of a verified implementation of the Motzkin polynomial.

8.1. Future Work

There are a range of questions left unexplored by the thesis, each of varying size and difficulty, including the following non exhaustive list:

- **Implementing a Set of Mutually Exclusive Fixed-Point Polynomials with Rational Coefficients** Chapter 4 considered only integer coefficients while subsequent chapters considered polynomials with rational coefficients. How best to merge such polynomials may also potentially lead to a question of how best to perform division by a set of mutually exclusive constants.

- **Delay or Power Optimising the Implementation of a Fixed-Point Polynomials** Where a measure of the hardware implementation cost has been required, least area or partial product bit count has been used throughout the thesis. The objective function used in Chapter 6 was the total number of partial product bits removes which may not necessarily correlate directly with area. The measures used enabled tractable optimisation problems to be posed and solved. However the approach required for optimising delay, power or a combination of all attributes may require a significant departure from those
used within these chapters.

- **Construct the Smallest Faithfully Rounded Multiplier with a Linear Compensation Function** The faithfully rounded multipliers created in Chapter 5 took existing architectures and found the conditions which guaranteed faithful rounding. In keeping with the approach taken in [135] an architecture could be derived from the need for faithful rounding as opposed to forcing faithful rounding conditions on a given architecture.

- **Constant Division via Truncated Arrays** The thesis contains a method for truncating an arbitrary binary array while maintaining an absolute error bound, however the method for constant division required a full, untruncated, multiply-add scheme. Methods should be explored for using truncation within such an array.

- **Provide Guidelines on the Level of Hardware Saving Achieved** For the various techniques presented in this thesis that exploit error freedom, it would be useful to provide to the user a statement on the degree of saving achieved and how sensitive these savings are to changes in the error specification. However this may be difficult to quantify. For example it is not possible to produce an infinitely precise result to a fixed-point division by three with finite output precision, therefore a measure of hardware saved is difficult to quantify. Providing a useful measure of sensitivity to changes in the error specification is also difficult. Where it may be possible to quantify the further number of partial product bits that could be saved if an error freedom were increased, it is difficult to express how this will translate into resultant hardware savings. Potentially, speculative synthesis experiments could be performed to provide the user with this information.

- **Polynomial Rewriting for Fixed-Point Polynomials with Rational Coefficients** Chapter 6 works on a given data flow graph of sum-of-products and constant division operations. Methods need to be constructed to automatically provide the best hardware implementation architecture.

- **Further Steps Towards Creating a Compiler for the Implementation of Floating-Point Polynomials** There are various as-
pects of Chapter 7 that could be automated. For example, a tool for automatic test pattern generation of inputs near the variety of a polynomial could be created to help test any potential implementation. Chapter 7 provides a concrete instance of the method but there remain many hurdles to creating a compiler capable of handling arbitrary polynomials.

8.2. Final Words

This thesis has explored how to design ASIC datapath that implements polynomials and where error freedom exists, how best to exploit it. In the case of no error freedom, this thesis has demonstrated an optimisation for the implementation of a set of mutually exclusive fixed-point polynomials. Where error freedom is present, industrial adoption of error exploiting or lossy techniques requires guarantees over the nature of the error introduced. The thesis has contributed to this goal by performing the mathematical analysis required to enable the creation of lossy components for multiplication, multioperand addition and constant division, all with guaranteed bounded maximum absolute error and with a significant reduction in hardware implementation costs when compared to correctly rounded equivalents. An approach to implementing a fixed-point polynomial with rational coefficients has been presented using these lossy components. Finally, a concrete example of how a particular floating-point polynomial can be implemented with bounded relative error has been presented and contributions towards the general case have been made.

Successful lossy synthesis requires guarantees on the nature of the introduced error and full exploitation of the error to minimise hardware implementation costs. This thesis has made significant steps in both of these areas and it is hoped that these advances will be built upon in the future.
Bibliography


[101] H.-J Ko and S.-F Hsiao. Design and Application of Faithfully Rounded and Truncated Multipliers with Combined Deletion, Reduction, Trun-


Appendices
A. Error Expectation and 
Variance of the CCT Multiplier 
Scheme

This appendix is associated with Chapter 5 and the introduction of truncated multiplier schemes. Chapter 5 focused on the determining the maximum absolute worst case error for a variety of truncated multiplier schemes. This appendix establishes the expectation and variance of the error associated with a particular truncated multiplier scheme, namely that of constant correction truncated multiplication (CCT).

A.1. Error Expectation

In order to determine the expected error of the scheme it will be assumed that the inputs to the multiplier are uniformly distributed. Recall that the error is defined as:

\[ \varepsilon = \frac{1}{2^n} \left( (AB + C2^k - \triangle_k) \mod 2^n - C2^k + \triangle_k \right) \]

The most difficult part of this calculation is determining the expectation of \( T \):

\[ T(A, B, C) = (AB + C2^k - \triangle_k) \mod 2^n \]

A.1.1. Lemma 2

\[ \min_{B} \{ T(A, B, 0) : T(A, B, 0) > 0, A > 0 \} = \max \left( 2^k, gcd(A, 2^n) \right) \]
Where $gcd(x, y)$ is the greatest common divisor of $x$ and $y$. Let us denote a value of $B$ that attains the minimum as $B^\dagger$. In addition define $g = \max(2^k, gcd(A, 2^n))$, i.e. $T(A, B^\dagger, 0) = g$.

**Proof**

Assume first $gcd(A, 2^n) > 2^k$ for $A > 0$.
In this case $T(A, B, 0) = AB \mod 2^n$.
Note that $A/gcd(A, 2^n)$ is odd and hence coprime to $2^n$.
Hence there exists $B^*$ such that $AB^*/gcd(A, 2^n) \equiv 1 \mod 2^n$.
Hence $AB^* \equiv gcd(A, 2^n) \mod 2^n$.
But note that $gcd(A, 2^n)$ divides $T(A, B, 0)$.
We may conclude that $gcd(A, 2^n)$ is the smallest value of $T$ that is possible.
Hence $\min_B\{T(A, B, 0) : T(A, B, 0) > 0, A > 0\} = gcd(A, 2^n)$.

Otherwise, let us assume that $gcd(A, 2^n) \leq 2^k$ for $A > 0$.
Note that $A/gcd(A, 2^n)$ is odd and hence coprime to $2^{n-k}$.
Hence there exists $B^*$ such that $AB^*/gcd(A, 2^n) \equiv 1 \mod 2^{n-k}$.
Hence $2^kAB^*/gcd(A, 2^n) \equiv 2^k \mod 2^n$.
Note that in the case when $gcd(A, 2^n) \leq 2^k$ and $B = 2^kB^*/gcd(A, 2^n)$ then $T(A, B, 0) = AB \mod 2^n$.
We may conclude that $2^k$ is the smallest value of $T$ that is possible.
Hence $\min_B\{T(A, B, 0) : T(A, B, 0) > 0, A > 0\} = 2^k$.

Combining the two cases proves the result. Note that in both cases: $\triangle_k(A, B^\dagger) = 0$.

**A.1.2. Statistics of $T$**

**Distribution of $T$ for fixed $A$**

It will now be shown that the distribution of $T(A, B, C)$ is uniform for fixed $A$.

Let $S(R) = \{B : T(A, B, C) = R\}$
In the following derivation Lemma 2 and the fact that $\Delta_k(A, B^\dagger, 0) = 0$ is used.

Hence $T(A, B + B^\dagger, C) = (T(A, B, C) + T(A, B^\dagger, 0)) \mod 2^n$.

Consider the set:

$$\left\{ \left( B + B^\dagger \right) \mod 2^n : T(A, B, C) = R \right\}$$

$$= \left\{ B : T(A, B - B^\dagger, C) = R \right\}$$

$$= \left\{ B : T(A, B, C) - T(A, B^\dagger, 0) = R \mod 2^n \right\}$$

$$= \left\{ B : T(A, B, C) - g = R \mod 2^n \right\}$$

$$= S ((R + g) \mod 2^n)$$

Hence incrementing each element in $S(R)$ by $B^\dagger$ modulo $2^n$ results in $S ((R + g) \mod 2^n)$. One can move between sets but can one move between all non zero sets by repeatedly adding $B^\dagger$ to elements? To answer this consider how close the residues can be, assuming $A > 0$:

$$\min_{B_1, B_2} \{(T(A, B_1, C) - T(A, B_2, C)) \mod 2^n : T(A, B_1, C) \neq T(A, B_2, C)\}$$

$$= \min_B \{ T(A, B, 0) : T(A, B, 0) > 0, \Delta_k = 0 \}$$

$$= \min_B \{ T(A, B, 0) : T(A, B, 0) > 0 \}$$

$$= g$$

Hence the minimal distance between residues is the same as that attained by adding $B^\dagger$ to $S(R)$. We may conclude that the sets $S(R)$ are of all equal sizes and are equally spaced. All residues are of the form:

$$(T(A, 0, C) + ig) \mod 2^n = \left( C 2^k + ig \right) \mod 2^n \quad i \in \left[0, \frac{2^n}{g} - 1 \right]$$

We may conclude that the distribution of $T(A, B, C)$ for fixed $A > 0$ assuming B is uniformly distributed as:

$$\mathbb{P} (T(A, B, C) = R|A) = \begin{cases} \frac{g}{2^n} & \text{if } g \mid (R - C 2^k) \\ 0 & \text{otherwise} \end{cases}$$
Expectation of T for fixed A

Assuming $A > 0$ and $g = \max (2^k, \gcd (A, 2^n))$:

\[
E(T(A, B, C)|A) = \frac{g}{2^n} \sum_{i=0}^{2^n/g-1} \left( C2^k + ig \right) \mod 2^n
\]

\[
= \frac{2^n - g}{2} + C2^k - g \left\lfloor \frac{C2^k}{g} \right\rfloor
\]

\[
= \frac{2^n - g}{2} + (C2^k \mod g)
\]

(A.1)

Expectation of T

Having computed the expectation of $T$ for fixed $A$, the general case can be determined as follows (recalling $g = \max (2^k, \gcd (A, 2^n))$):

\[
E(T(A, B, C)) = \sum_{A=0}^{2^n-1} E(T(A, B, C)|A)P(A)
\]

\[
= \frac{1}{2^n} \sum_{A=0}^{2^n-1} E(T(A, B, C)|A)
\]

\[
= \frac{1}{2^n} \left( E(T(A, B, C)|A = 0) + \sum_{A=1}^{2^n-1} \left( \frac{2^n - g}{2} + C2^k \mod g \right) \right)
\]

Now note that the number of numbers $p$ between 0 and $2^n$ with $\gcd(p, 2^n) = 2^q$ are binary numbers of the form \(XX..XX100..00\), where $X$ is a don’t care bit, and as such there are $2^{n-q-1}$ such numbers. Using this and summing over the the various values of $i = \gcd(a, 2^n)$:

\[
E(T(A, B, C)) = \frac{1}{2^n} \left( C2^k + \sum_{i=0}^{n-1} 2^{n-i-1} \left( \frac{2^n - \max (2^k, 2^i)}{2} + C2^k \mod \left( \max (2^k, 2^i) \right) \right) \right)
\]

Which after some manipulation becomes:

\[
E(T(A, B, C)) = \frac{2^n - 2^k}{2} - \frac{n - k}{4} + \frac{\text{Hamm}(C)}{2}
\]

(A.2)
Where $\text{Hamm}(C)$ is the Hamming weight of $C$, i.e. the number of non zero bits in its binary form.

**Expectation of $\triangle_k$**

For completeness the following expectation is needed:

$$
\mathbb{E}(\triangle_k) = \mathbb{E}
\left(\sum_{i=0}^{k-1} \sum_{j=0}^{k-i-1} a_ib_j2^{i+j}\right)
$$

$$
= \mathbb{E}(a_0)\mathbb{E}(b_0)\sum_{i=0}^{k-1} 2^i \left(2^{k-i} - 1\right)
$$

$$
= \frac{1}{4} \left((k - 1)2^k + 1\right)
$$

(A.3)

**The Expected Error**

The expectation of the error can now be stated:

$$
\mathbb{E}(\varepsilon) = \frac{1}{2n}\mathbb{E}\left(T(A,B,C) - C2^k + \triangle_k\right)
$$

Using (A.2) and (A.3) this becomes:

$$
\mathbb{E}(\varepsilon) = \frac{1}{2} + \frac{1}{2^{n+2}} \left(2^k(k - 4C - 3) - n + k + 2\text{Hamm}(C) + 1\right)
$$

(A.4)

Now it is useful to note that (A.4) is strictly monotonically decreasing in $C$. To see this, assume without loss of generality that $C$ can be written uniquely as $C = (2X + 1)\gcd(C, 2^n)$ for $X \in \mathbb{N}$, note that this notation is simply stating that the binary expansion of $C$ is of the form $X100..00$. Then the function $f$, below, is strictly increasing and hence (A.4) is strictly monotonic.
decreasing in $C$:

$$\mathbb{E}(z) = \frac{1}{2} + \frac{1}{2^{n+2}} \left( 2^k(k - 3) - n + k + 1 - 4f(C) \right)$$

Where \( f(C) = C2^k - \frac{1}{2}\text{Hamm}(C) \)

\[
\begin{align*}
  f(C) - f(C - 1) &= 2^k + \frac{1}{2} (\text{Hamm}(C - 1) - \text{Hamm}(C)) \\
  &= 2^k + \frac{1}{2} (\text{Hamm}(X) + \log_2 \gcd(C, 2^n) - \text{Hamm}(X) - 1) \\
  &= 2^k - \frac{1}{2} + \frac{1}{2} \log_2 \gcd(C, 2^n) \\
  &> 0 \quad \text{as} \quad k \in \mathbb{N}
\end{align*}
\]
A.2. Error Variance

In order to determine the error variance of the scheme, recall the error definition:

\[ \varepsilon = \frac{1}{2^n} \left( T(A, B, C) - C2^k + \triangle_k \right) \]

\[ \text{Var}(\varepsilon) = \frac{1}{2^{2n}} \text{Var}(T(A, B, C) + \triangle_k) \]

\[ = \frac{1}{2^{2n}} \left( \mathbb{E}(T(A, B, C) + \triangle_k)^2 - \mathbb{E}^2(T(A, B, C) + \triangle_k) \right) \]

\[ = \frac{1}{2^{2n}} \left( \mathbb{E}(T(A, B, C))^2 + \mathbb{E}(\triangle_k)^2 + 2\mathbb{E}(T(A, B, C)\triangle_k) - (\mathbb{E}(T(A, B, C)) + \mathbb{E}(\triangle_k))^2 \right) \]  

(A.5)

Picking off each term in turn:

A.2.1. \( \mathbb{E}(T^2(A, B, C)) \)

The expectation of \( T^2 \) given a fixed value of \( A \) is first computer, in a similar fashion to (A.1). Again assuming \( A > 0 \) and \( g = \max(2^k, \gcd(A, 2^n)) \).

After much algebraic simplification:

\[ \mathbb{E}(T^2(A, B, C)|A) = \frac{g}{2^n} \sum_{i=0}^{2^n/g-1} \left( \left( C2^k + ig \right) \mod 2^n \right)^2 \]

\[ = \frac{1}{6} \left( 2^n - g \right) \left( 2^{n+1} - g + 6 \left( \left( C2^k \right) \mod g \right) \right) \]

(A.6)
Then proceeding to compute the expectation of $T^2$, note that $C[j - 1 : 0]$ indicates the least significant $j$ bits of $C$:

$$
\mathbb{E}(T^2(A, B, C)) = \frac{1}{2^n} \sum_{A=0}^{2^n-1} \mathbb{E}(T^2(A, B, C)|A)
$$

$$
= \frac{1}{2^n} \left( C^2 2^{2k} + \sum_{A=1}^{2^n-1} \left( \frac{1}{6} (2^n - g) \left( 2^{n+1} - g + 6 \left( \left( C2^k \right) \mod g \right) \right) + \left( \left( C2^k \right) \mod g \right)^2 \right) \right)
$$

$$
= \frac{1}{12} \left( 2^n - 2^k \right) \left( 2^{n+2} - 2^{k+1} + 3 \right) - \frac{1}{4} (n - k) \left( 2^n + C2^{k+1} \right)
$$

$$
+ 2^{n-1} \text{Hamm}(C) + 2^k \sum_{j=1}^{n-k-1} C_j \left( C[j - 1 : 0] + j2^{j-1} \right)
$$

(A.7)

A.2.2. $\mathbb{E} (\triangle_k^2)$

This expectation is relatively simple, using the Kronecker delta $\delta_{ij}$ which is defined as:

$$
\delta_{ij} = \begin{cases} 
1 & i = j \\
0 & i \neq j
\end{cases}
$$

Using this notation the expectation can be expressed as follows:

$$
\mathbb{E} (\triangle_k^2) = \mathbb{E} \left( \sum_{i+j<k} a_i b_j 2^{i+j} \right)^2
$$

$$
= \sum_{i+j<k} \sum_{p+q<k} \mathbb{E} (a_i a_p) \mathbb{E} (b_j b_q) 2^{i+j+p+q}
$$

$$
= \frac{1}{16} \sum_{i+j<k} \sum_{p+q<k} (1 + \delta_{ip})(1 + \delta_{jq}) 2^{i+j+p+q}
$$

$$
= \frac{1}{144} \left( 4^k \left( 9k^2 + 3k - 22 \right) + 18(k+1)2^k + 4 \right)
$$

(A.8)

A.2.3. $\mathbb{E} (T(A, B, C)\triangle_k)$

This cross term is of more interest:

$$
\mathbb{E} (T(A, B, C)\triangle_k) = \sum_{a=0}^{2^k-1} \mathbb{E} \left( T\triangle_k|A \mod 2^k = a \right) \mathbb{P} \left( A \mod 2^k = a \right)
$$

321
Now if \( A \mod 2^k = 0 \) then \( \triangle_k = 0 \) hence:

\[
E\left( T(A, B, C) \triangle_k \mid A \mod 2^k = 0 \right) = 0
\]

Continuing with the expectation:

\[
E (T(A, B, C) \triangle_k) = \\
\frac{1}{2^k} \sum_{a=1}^{2^{k-1}} \sum_{t=0}^{2^{n-k}-1} E \left( T \triangle_k \mid T = t2^k, A \mod 2^k = a \right) P \left( T = t2^k \mid A \mod 2^k = a \right)
\]

Now recall for \( A > 0 \) and \( g = \max (2^k, \gcd(a, 2^n)) \):

\[
P \left( T(A, B, C) = R \mid A \right) = \\
\begin{cases} 
\frac{g}{2^n} & \text{if } g \mid (R - C2^k) \\
0 & \text{otherwise}
\end{cases}
\]

In the current calculation \( A \mod 2^k > 0 \) hence \( \gcd(a, 2^n) < 2^k \), hence \( g = 2^k \) and:

\[
P \left( T(A, B, C) = t2^k \mid A \mod 2^k = a > 0 \right) = \frac{1}{2^{n-k}}
\]

Hence

\[
E (T(A, B, C) \triangle_k) = \frac{1}{2^{n-k}} \sum_{a=1}^{2^{k-1}} \sum_{t=0}^{2^{n-k}-1} t E \left( \triangle_k \mid T = t2^k, A \mod 2^k = a \right)
\]

The expectation of \( \triangle_k \) is independent of \( T(A, B, C) \) for fixed \( A \) and \( g = 2^k \). To see this, consider the set of all possible values that \( \triangle_k \) can take given a value of \( T(A, B, C) \). Recall from Lemma 2 that there exists \( B^\dagger \) such that
\( T(A, B^\dagger, 0) = g \) and satisfies \( \triangle_k(A, B^\dagger) = 0 \):

\[
M(t) = \{ \triangle_k(A, B) \mid T(A, B, C) = t2^k \mod 2^n \}
\]

\[
M(t + 1) = \{ \triangle_k(A, B) \mid T(A, B, C) = (t + 1)2^k \mod 2^n \}
\]

\[
= \{ \triangle_k(A, B) \mid T(A, B, C) = g = t2^k \mod 2^n \}
\]

\[
= \{ \triangle_k(A, B) \mid T(A, B, C) = T(A, B^\dagger, 0) = t2^k \mod 2^n \}
\]

\[
= \{ \triangle_k(A, B) \mid T(A, B^\dagger, C) = t \}
\]

\[
= \{ \triangle_k(A, B) \mid T(A, B, C) = t \}
\]

\[
= M(t)
\]

So the set of possible values for fixed \( A \) that \( \triangle_k(A, B) \) is independent of \( T(A, B, C) \), we may conclude that

\[
\mathbb{E} \left( \triangle_k \middle| T = t2^k, A \mod 2^k = a \right) = \mathbb{E} \left( \triangle_k \mid A \mod 2^k = a \right)
\]

Continuing with the expectation at hand:

\[
\mathbb{E} \left( T(A, B, C) \triangle_k \right) = \frac{1}{2^{n-k}} \sum_{a=1}^{2^n-1} \sum_{t=0}^{2^{n-k}-1} t \mathbb{E} \left( \triangle_k \middle| T = t2^k, A \mod 2^k = a \right)
\]

\[
= \frac{1}{2^{n-k}} \left( \sum_{t=0}^{2^{n-k}-1} t \right) \sum_{a=1}^{2^{n-k}-1} \mathbb{E} \left( \triangle_k \mid A \mod 2^k = a \right)
\]

\[
= \frac{1}{2} \left( 2^{n-k} - 1 \right) \sum_{a=1}^{2^{n-k}-1} \left( \sum_{i+j<k} a_i b_j 2^{i+j} \mid a \right)
\]

\[
= \frac{1}{4} \left( 2^{n-k} - 1 \right) \sum_{a=1}^{2^{n-k}-1} \sum_{i+j<k} a_i 2^{i+j}
\]

\[
= \frac{1}{8} \left( 2^n - 2^k \right) \left( (k - 1)2^k + 1 \right) \quad (A.9)
\]
A.2.4. The Error Variance

Now inserting (A.7), (A.8), (A.9), (A.2) and (A.3) into (A.5) and simplifying:

\[
\text{Var}(\varepsilon) = \frac{1}{12} + \frac{1}{2n+2} + \frac{21k - 43}{9 \times 4^{n-k+2}} - \frac{5}{9 \times 4^{n+2}} \\
+ \frac{n - k}{4^{n+2}} \left( (k - 4C - 3)2^{k+1} - n + k + 4\text{Hamm}(C) + 2 \right) \\
+ \frac{1}{2^{2n-k}} \sum_{j=1}^{n-k-1} C_j \left( C[j-1 : 0] + j2^{j-1} \right) \\
- \frac{\text{Hamm}(C)}{4^{n+1}} \left( \text{Hamm}(C) + (k - 3)2^k + 1 \right)
\]  

(A.10)

This concludes the appendix in which the analytic formulae for the expectation and variance of the error associated with the constant correction truncated multiplier.
B. Construction of Faithfully Rounded Floating-Point Addition

This appendix is associated with Chapter 7 and described how a faithfully rounded floating-point adder can be constructed, the architecture presented here exists within the industry tool CellMath Designer [57]. The implementation, along with the proof that it does perform a faithful rounding, will first be presented and then the results of logic synthesis will demonstrate the value of such a component over correctly rounded floating-point adders.

In the case of floating-point addition, the complexities of implementing correct rounding can be greatly simplified by permitting faithful rounding. Consider adding \( a \) and \( b \) which are assumed to be non-exceptional normal numbers:

\[
y = a + b = (1)^{sa}2^{expa-bias}1.manta + (1)^{sb}2^{expb-bias}1.mantb \\
= (1)^{sa}2^{expa-bias}(1.manta + (1)^{sa+sb}2^{expb-expa}1.mantb) \\
= (1)^{sa}2^{expa-bias}(1.manta + (1)^{sa+sb}(1.mantb >> (expa - expb)))
\]

In order to be faithfully rounded, the required precision of the following term needs to be ascertained:

\[
mantprenorm = 1.manta + (1)^{sa+sb}(1.mantb >> (expa - expb))
\]
Case 1: \( expa = expb, sa = sb \)

The mantissa calculation reduces to:

\[
\text{mantprenorm}[mw + 1 : 0] = 1\cdot\text{manta}[mw - 1 : 0] + 1\cdot\text{mantb}[mw - 1 : 0]
\]

This will be faithfully rounded if the output mantissa \( \text{manty} \) is taken to be \( \text{mantprenorm}[mw : 1] \).

Case 2: \( expa = expb, sa \neq sb \)

In this case the mantissa calculation reduces to:

\[
\text{mantprenorm}[mw + 1 : 0] = 1\cdot\text{manta}[mw - 1 : 0]\cdot\text{mantb}[mw - 1 : 0]
\]

In this case, if the full answer is kept and is renormalised, no rounding is required.

Case 3: \( expa > expb, sa = sb \)

In this case the mantissa calculation reduces to:

\[
\text{mantprenorm}[mw + 1 : 0] = 1\cdot\text{manta}[mw - 1 : 0] + (1\cdot\text{mantb}[mw - 1 : 0] >> (expa - expb))
\]

In this case some of the bits of \( \text{mantb} \) have already shifted out, but the leading one is either \( \text{mantprenorm}[mw + 1] \) or \( \text{mantprenorm}[mw] \) so this will still meet the faithfully rounded condition. The final result for \( \text{manty} \) is:

\[
\text{manty} = \text{mantprenorm}[mw + 1]? \text{mantprenorm}[mw : 1] : \text{mantprenorm}[mw - 1 : 0]
\]

326
Case 4: $expa > expb$, $sa \neq sb$

In this case the mantissa calculation reduces to:

$$\text{mantprenorm}[mw + 1 : 0] = 1.manta[mw - 1 : 0]$$

$$(1.mantb[mw - 1 : 0] \gg (expa - expb))$$

However this equation, as presented, does not keep enough bits to return a faithfully rounded result. For example $expa - expb = 1$, $manta = 0$ and $mantb = 1$. In this case $mantprenorm = 1 << (mw - 1)$ and $manty = 0$, however the correct answer is one unit in the last place less than this, thus violating the faithfully rounded condition. In order to mitigate this situation, one guard bit can be added, as so:

$$\text{mantprenorm}[mw + 2 : 0] = (1.manta[mw - 1 : 0] << 1)$$

$$((1.mantb[mw - 1 : 0] << 1) \gg (expa - expb))$$

Conclusion

So the algorithm is (assuming $|a| \geq |b|$, if not, swap operands) is:

$$ta[mw + 1 : 0] = (1 << (mw + 1)) + (manta[mw - 1 : 0] << 1)$$

$$tb[mw + 1 : 0] = (1 << (mw + 1)) + (mantb[mw - 1 : 0] << 1)$$

$$\text{shiftb}[mw + 10] = tb >> (expa - expb)$$

$$\text{mantprenorm}[mw + 2 : 0] = (sa == sb)? \ ta + \text{shiftb} : \ tashiftb$$

$$\text{norm}[mw + 2 : 0] = \text{mantprenorm} << \text{leadingzeroes}(\text{mantprenorm})$$

$$\text{manty}[mw - 1 : 0] = \text{norm}[mw + 1 : 2]$$

Where $\text{leadingzeroes}(x)$ returns the number of leading zeroes in $x$.

Experiments

To see the benefits of these faithfully rounded floating-point adders, two architectures were constructed, one following the architecture above and the other using a two path adder approach. These were synthesised against reference designs from Synopsys’ DesignWare library [163] for various rounding
modes. Comparisons were performed for adders with F16 and F32 inputs. The area-delay curves which resulted from performing logic synthesis with Synopsys’ Design Compiler can be found in Figures B.1 and B.2. These figures contain DesignWare round to nearest, ties to even (dw_rte) and round towards zero (dw_rtz) adders and the two faithfully rounded architectures (img_fr1_arch1 and img_fr1_arch2). The other designs are attempts to improve upon the rte and rtz designs. The results show that F16 faithfully rounded adders can be 48% faster then DesignWare and up to 73% smaller. The results show that F32 faithfully rounded adders can be 43% faster then DesignWare and up to 60% smaller.

Figure B.1.: Area Delay Curves for F16 Adder Architectures.
Figure B.2.: Area Delay Curves for F32 Adder Architectures.
C. The Condition for the Use of F32/F64 Far from $x = y = z$

This appendix is associated with Chapter 7 and in particular the accurate evaluation of the Motzkin polynomial. Accurate evaluation far from the variety requires less precision and this appendix determines the region where F32 and F64 can be used.

C.1. The Conditions for the Use of F32

It is required to calculate, for $p = z^6 + x^2y^2(x^2 + y^2 - 3z^2)$:

$$\min_{e > 0} \left( \frac{6 \left(1 + 2^{-23}\right)^8 - 1}{\min_{p(x)} \left( |x| = 1, x, y, z \geq 0, x \leq 2y, y \leq 2x, x \leq 2z, y \leq 2z, \right)} \right) < 2^{-10}$$

Rearranging and simplifying:

$$\min_{e > 0} \left( 3 \times 2^{11} \left(1 + 2^{-23}\right)^8 - 1 \right) < \min_{p(x)} \left( |x| = 1, x, y, z \geq 0, x \leq 2y, y \leq 2x, x \leq 2z, y \leq 2z, \right)$$

Taking the first two terms of the expansion of $(1 + 2^{-23})^8$ and rounding up will give a conservative estimate on $e$:

$$\min_{e > 0} \left( \frac{201326679}{34359738368} \right) \leq \min_{p(x)} \left( |x| = 1, x, y, z \geq 0, x \leq 2y, y \leq 2x, x \leq 2z, y \leq 2z, \right)$$
Now the minimum of \( p \) will occur closest to the line \( x = y = z \), so without loss of generality assume \( |y - z| = e \) as all the equations are symmetric in \( x \) and \( y \):

\[
\min_{e > 0} \left( \frac{201326679}{34359738368} \leq \begin{cases} 
    x^2 + y^2 + z^2 = 1, x, y, z \geq 0, \\
    p(x) : x \leq 2y, y \leq 2x, x \leq 2z, y \leq 2z, \\
    (|x - z| \geq e \text{ or } |y - z| = e)
\end{cases} \right)
\]

Now consider the points which satisfy:

\[
p = \frac{201326679}{34359738368} \quad x^2 + y^2 + z^2 = 1 \quad y - z = e \quad (C.1)
\]

Substituting the latter conditions into \( p \) gives:

\[
p = 9z^6 + 24z^5e + 28z^4e^2 - 6z^4 - 16z^3e^3 - 14z^3e + 4z^2e^4
- 11z^2e^2 + z^2 - 4ze^3 + 2ze - e^4 + e^2
\]

The minimum of this expression will occur when \( \partial p/\partial z = 0 \), namely:

\[
54z^5 + 120z^4e + 112z^3e^2 - 24z^3 + 48z^2e^3 - 42z^2e + 8ze^4
- 22ze^2 + 2z - 4e^3 + 2e = 0 \quad (C.2)
\]

Together, the equations in C.1 and C.2, are four equations in four unknowns. These equations can be solved using Gröbner bases. The solutions which have \( x, y, z \geq 0 \) are:

\[
x = 0.66720195 \quad y = 0.4560641 \quad z = 0.58894322 \quad e = -0.13288681
x = 0.47164497 \quad y = 0.69173132 \quad z = 0.54686268 \quad e = 0.14486864
\]

Note that both of these solutions satisfy the other required conditions \( x \leq 2y, y \leq 2x, x \leq 2z \) and \( y \leq 2z \). It can be confirmed that these points are both minima of \( p \) by inspecting \( \partial^2 p/\partial z^2 \). The first solution corresponds to a minimum on \( y - z = -e \) and the latter to \( y - z = e \). This has investigated the turning points on the curved pyramid. The corners of the curved pyramid would need to satisfy:

\[
p = \frac{201326679}{34359738368} \quad x^2 + y^2 + z^2 = 1 \quad y - z = \pm e \quad x - z = \pm e
\]

331
The solutions from these sets of equations are:

\[
\begin{align*}
  x &= 0.55481331 & y &= 0.55481331 & z &= 0.61997127 & e &= -0.065157963 \\
  x &= 0.59903995 & y &= 0.59903995 & z &= 0.53132126 & e &= 0.067718682 \\
  x &= 0.44690999 & y &= 0.69052864 & z &= 0.56871931 & e &= -0.12180932 \\
  x &= 0.69052864 & y &= 0.44690999 & z &= 0.56871931 & e &= 0.12180932
\end{align*}
\]

All of these solutions the largest value of \(|e|\) is \(e = 0.14486864\), so if a curved pyramid with this ratio is maintained, then \(p \geq \frac{201326679}{34359738368}\).

Now the exact value of this particular \(e\) is rounded up to a representable F32:

\[
e_1 = \frac{4860985}{2^{25}} \approx 0.1448686421
\]

The Singular code used to perform these operations is below.

```singular
ring r=0,(x,y,z,e),lp;
poly p = z6+x2y2*(x2+y2-3z2);
ideal I = x2+y2+z2-1,y-z-e;
poly q = diff(reduce(p,groebner(I)),z);
ideal J = p-201326679/34359738368,I,q;
LIB "solve.lib";
def R = solve(groebner(J));
setring R;
SOL[12];
SOL[16];
```

```singular
ring r=0,(x,y,z,e),lp;
ideal I = z6+x2y2*(x2+y2-3z2)-201326679/34359738368,x2+y2+z2-1,x-z-e,y-z-e;
LIB "solve.lib";
def R = solve(groebner(I));
setring R;
SOL[4];
SOL[6];
```

```singular
ring r=0,(x,y,z,e),lp;
ideal I = z6+x2y2*(x2+y2-3z2)-201326679/34359738368,x2+y2+z2-1,x-z-e,y-z+e;
```

332
LIB "solve.lib";
def R = solve(groebner(I));
setring R;
SOL[2];
SOL[4];

C.2. The Conditions for the Use of F64

Repeating the process for F64, it is required to calculate, for \( p = z^6 + x^2y^2(x^2 + y^2 - 3z^2) \) for doubles:

\[
\min_{e>0} \left( \frac{6 \left( (1 + 2^{-52})^8 - 1 \right)}{\min \left( p(x) : \begin{array}{l}
|x| = 1, x, y, z \geq 0, x \leq 2y, y \leq 2x, x \leq 2z, y \leq 2z, \\
((x - z)^2 \geq e^2(x^2 + y^2 + z^2)) \text{ or } (y - z)^2 \geq e^2(x^2 + y^2 + z^2)
\end{array} \right)} \right) < 2^{-10}
\]

Rearranging and simplifying:

\[
\min_{e>0} \left( 3 \times 2^{11} \left( (1 + 2^{-52})^8 - 1 \right) \right) < \min \left( p(x) : \begin{array}{l}
|x| = 1, x, y, z \geq 0, x \leq 2y, y \leq 2x, x \leq 2z, y \leq 2z, \\
\left| x - z \right| \geq e \text{ or } \left| y - z \right| \geq e
\end{array} \right)
\]

Taking the first two terms of the expansion of \((1 + 2^{-52})^8\) and rounding up will give a conservative estimate on \( e \):

\[
\min_{e>0} \left( \frac{108086391056891991}{2^{93}} \right) \leq \min \left( p(x) : \begin{array}{l}
x^2 + y^2 + z^2 = 1, x, y, z \geq 0, x \leq 2y, y \leq 2x, x \leq 2z, y \leq 2z, \\
\left| x - z \right| \geq e \text{ or } \left| y - z \right| \geq e
\end{array} \right)
\]

Now the minimum of \( p \) will occur closest to the line \( x = y = z \), so without loss of generality assume \( |y - z| = e \) as all the equations are symmetric in \( x \) and \( y \):

\[
\min_{e>0} \left( \frac{108086391056891991}{2^{93}} \right) \leq \min \left( p(x) : \begin{array}{l}
x^2 + y^2 + z^2 = 1, x, y, z \geq 0, x \leq 2y, y \leq 2x, x \leq 2z, y \leq 2z, \\
\left| x - z \right| \geq e \text{ or } \left| y - z \right| = e
\end{array} \right)
\]

333
Now consider the points which satisfy:

\[ p = \frac{108086391056891991}{2^{93}} \quad x^2 + y^2 + z^2 = 1 \quad y - z = e \quad (C.3) \]

Substituting the latter conditions into \( p \) gives:

\[
p(z, e) = 9z^6 + 24z^5e + 28z^4e^2 - 6z^4 + 16z^3e^3 - 14z^3e + 4z^2e^4
- 11z^2e^2 + z^2 - 4ze^3 + 2ze - e^4 + e^2
\]

The minimum of this expression will occur when \( \partial p/\partial z = 0 \), namely:

\[
54z^5 + 120z^4e + 112z^3e^2 - 24z^3 + 48z^2e^3 - 42z^2e + 8ze^4
- 22ze^2 + 2z - 4e^3 + 2e = 0 \quad (C.4)
\]

Together, the equations in C.3 and C.4, are four equations in four unknowns. These equations can be solved using Gröbner bases. The solutions which have \( x, y, z \geq 0 \) are:

\[
x = 0.5773464545 \quad y = 0.5773550376 \quad z = 0.5773493155 \quad e = 0.000005722055350
\]

Note that this solution satisfies the other required conditions \( x \leq 2y, y \leq 2x, x \leq 2z \) and \( y \leq 2z \). It can be confirmed that this point is a minimum of \( p \) by inspecting \( \partial^2 p/\partial z^2 \). This has investigated the turning points on the curved pyramid. The corners of the curved pyramid would need to satisfy:

\[
p = \frac{108086391056891991}{2^{93}} \quad x^2 + y^2 + z^2 = 1 \quad y - z = \pm e \quad x - z = \pm e
\]

The solutions from these sets of equations are:

\[
x = 0.57734932 \quad y = 0.57734932 \quad z = 0.57735218 \quad e = -0.0000028610206
x = 0.57735122 \quad y = 0.57735122 \quad z = 0.57734836 \quad e = 0.0000028610253
x = 0.57734531 \quad y = 0.57735522 \quad z = 0.57735027 \quad e = -0.0000049554371
x = 0.57735522 \quad y = 0.57734531 \quad z = 0.57735027 \quad e = 0.0000049554371
\]

All of these solutions the largest value of \( |e| \) is \( e = 0.000005722055350 \), so if a curved pyramid with this ratio is maintained, then \( p \geq 108086391056891991/2^{93} \). Now the exact value of this particular \( e \) is rounded up to a representable
The Maple code used to perform these operations is below (note that the use of Maple versus Singular rests on the size of integer constants supported by the respective tools and runtimes of some of the calculations).

This provides the equation that $e$ satisfies:

```plaintext
SemiAlgebraic({x > 0, y > 0, z > 0, -z+y = e, x^2+y^2+z^2 = 1, y^2*x^4+y^4*x^2-3*x^2*y^2*z^2+z^6-108086391056891991/2^93 = 0, 54*z^5+120*e*z^4+112*e^2*z^3-24*z^3+48*e^3*z^2-42*e*z^2+8*e^4*z -22*e^2*z+2*z-4*e^3+2*e = 0}, [e, x, y, z])
```

This finds the positive root of $e$:

```plaintext
fsolve({-z+y = e, x^2+y^2+z^2 = 1, y^2*x^4+y^4*x^2-3*x^2*y^2*z^2+z^6-108086391056891991/2^93 = 0, 54*z^5+120*e*z^4+112*e^2*z^3-24*z^3+48*e^3*z^2-42*e*z^2+8*e^4*z -22*e^2*z+2*z-4*e^3+2*e = 0}, {e, x, y, z}, 0 .. 1)
```

This finds the positive root of $-e$, but there is none:

```plaintext
fsolve({-z+y = -e, x^2+y^2+z^2 = 1, y^2*x^4+y^4*x^2-3*x^2*y^2*z^2+z^6-108086391056891991/2^93 = 0, 54*z^5-120*e*z^4+112*e^2*z^3-24*z^3-48*e^3*z^2+42*e*z^2+8*e^4*z -22*e^2*z+2*z+4*e^3-2*e = 0}, {e, x, y, z}, 0 .. 1)
```

```plaintext
ring r=0,(x,y,z,e),lp; ideal I = z6+x2*y2*z2-3*x2*y2-2*z2-1, x2+y2+z2-1, x-z-e, y-z-e; LIB "solve.lib"; def R = solve(groebner(I)); setring R; SOL[4]; SOL[6];

ring r=0,(x,y,z,e),lp;
```

335
ideal I = z6+x2y2*(x2+y2-3z2)
-108086391056891991/9903520314283042199192993792,
x2+y2+z2-1,x-z-e,y-z+e;
LIB "solve.lib";
def R = solve(groebner(I));
setring R;
SOL[2];
SOL[4];
D. The Condition for the Use of F32/F64 Far from $z = x = 0$

This appendix is associated with Chapter 7 and in particular the accurate evaluation of the Motzkin polynomial. Accurate evaluation far from the variety requires less precision and this appendix determines the region where F32 and F64 can be used.

D.1. The Conditions for the Use of F32

It is required to calculate, for $p = z^6 + x^2y^2(x^2 + y^2 - 3z^2)$:

$$\min_{e > 0} \left( \frac{6 \left(1 + 2^{-23}\right)^8 - 1}{\min \left( \frac{\sum_{i=1}^{10} 2x \leq y, 2z \leq y, (x \geq e \text{ or } z \geq e)}{p(x) : ||x|| = 1, x, y, z \geq 0, 2x \leq y, 2z \leq y, (x \geq e \text{ or } z \geq e)} \right)} < 2^{-10} \right)$$

Rearranging and simplifying:

$$\min_{e > 0} \left( 3 \times 2^{11} \left(1 + 2^{-23}\right)^8 - 1 \right) < \min \left( p(x) : ||x|| = 1, x, y, z \geq 0, 2x \leq y, 2z \leq y, (x \geq e \text{ or } z \geq e) \right)$$

Taking the first two terms of the expansion of $(1 + 2^{-23})^8$ and rounding up will give a conservative estimate on $e$:

$$\min_{e > 0} \left( \frac{20326679}{34359738368} \leq \min \left( p(x) : x^2 + y^2 + z^2 = 1, x, y, z \geq 0, 2x \leq y, 2z \leq y, (x \geq e \text{ or } z \geq e) \right) \right)$$
Now the minimum of $p$ will occur closest to the $y$ axis, so first assuming $x = e$:

$$\min_{c > 0} \left( \frac{201326679}{34359738368} \leq \left( p(x) : \begin{array}{l} x^2 + y^2 + z^2 = 1, x, y, z \geq 0, \\ 2x \leq y, 2z \leq y, \\ (x = e \text{ or } z \geq e) \end{array} \right) \right)$$

Now consider the points which satisfy:

$$p = \frac{201326679}{34359738368} x^2 + y^2 + z^2 = 1 \quad x = e \quad (D.1)$$

Substituting the latter conditions into $p$ gives:

$$p(z, e) = z^6 + 4z^4e^2 + 4z^2e^4 - 5z^2e^2 - e^4 + e^2$$

The minimum of this expression will occur when $\partial p/\partial z = 0$, namely:

$$6z^5 + 16z^3e^2 + 8ze^4 - 10ze^2 = 0 \quad (D.2)$$

Together, the equations in D.1 and D.2, are four equations in four unknowns. These equations can be solved using Gröbner bases. The variable $e$ must satisfy:

$$11682667931703362164193616740741216e^{14} - 29328364286880315433027725359579136e^{12} + 25576118536498892981215668421525504e^{10} - 8987880450068701936858640347561984e^8 + 1081419846482511806373812062126080e^6 - 18863893044946249761968854401024e^4 + 112807383575555458701226672128e^2 - 220327016256688764945910653 = 0$$

The solutions which have $x, y, z \geq 0$ are:

$$x = 0.07677316 \quad y = 0.99704859 \quad z = 0 \quad e = 0.07677316$$
$$x = 0.99704859 \quad y = 0.07677316 \quad z = 0 \quad e = 0.99704859$$
$$x = 0.095825431 \quad y = 0.93765632 \quad z = 0.33409298 \quad e = 0.095825431$$
$$x = 0.44677538 \quad y = 0.68891201 \quad z = 0.57078192 \quad e = 0.44677538$$
$$x = 0.69570237 \quad y = 0.45495471 \quad z = 0.55589066 \quad e = 0.69570237$$

338
Only the first and third solutions satisfy $2x \leq y$ and $2z \leq y$. It can be confirmed that these points are both minima of $p$ by inspecting $\partial^2 p/\partial z^2$.

Repeating the exercise for $z = e$:

$$\min_{e > 0} \left( \frac{201326679}{34359738368} \leq \left( \begin{array}{c} x^2 + y^2 + z^2 = 1, x, y, z \geq 0, \\ 2x \leq y, 2z \leq y, \\ (x \geq e \text{ or } z = e) \end{array} \right) \right)$$

Now consider the points which satisfy:

$$p = \frac{201326679}{34359738368} \ x^2 + y^2 + z^2 = 1 \quad z = e \quad (D.3)$$

Substituting the latter conditions into $p$ gives:

$$q(x, e) = 4x^4e^2 - x^4 + 4x^2e^4 - 5x^2e^2 + x^2 + e^6$$

The minimum of this expression will occur when $\partial q/\partial x = 0$, namely:

$$16x^3e^2 - 4x^3 + 8xe^4 - 10xe^2 + 2x = 0 \quad (D.4)$$

Together, the equations in D.3 and D.4, are four equations in four unknowns. These equations can be solved using Gröbner bases. The variable $e$ must satisfy:

$$2656331146614175432704e^{10} - 1770887431076116955136e^8$$

$$+ 28823037316241450984e^6 - 15564447038111219712e^4$$

$$+ 10376298025407479808e^2 - 1688850572557410927 = 0$$

The solutions which have $x, y, z \geq 0$ are:

$$x = 0 \quad y = 0.90538322 \quad z = 0.42459536 \quad e = 0.42459536$$

$$x = 0.59903995 \quad y = 0.59903995 \quad z = 0.53132126 \quad e = 0.53132126$$

$$x = 0.55481331 \quad y = 0.55481331 \quad z = 0.61997127 \quad e = 0.61997127$$

Only the first solution satisfies $2x \leq y$ and $2z \leq y$. It can be confirmed that this point is a minimum of $p$ by inspecting $\partial^2 p/\partial z^2$.

This has investigated the turning points on the curved pyramid. The
corners of the curved pyramid would need to satisfy:

\[ p = \frac{201326679}{34359738368} \quad x^2 + y^2 + z^2 = 1 \quad (x, z) = (0, e), (e, 0), (e, e) \]

The solutions from these sets of equations are:

\[
\begin{align*}
  x &= 0.07677316 & y &= 0.99704859 & z &= 0 & e &= 0.07677316 \\
  x &= 0.99704859 & y &= 0.07677316 & z &= 0 & e &= 0.99704859 \\
  x &= 0 & y &= 0.90538322 & z &= 0.42459536 & e &= 0.42459536 \\
  x &= 0.077968503 & y &= 0.99390232 & z &= 0.077968503 & e &= 0.077968503 \\
  x &= 0.53440394 & y &= 0.6548472 & z &= 0.53440394 & e &= 0.53440394 \\
  x &= 0.61237244 & y &= 0.49999998 & z &= 0.61237244 & e &= 0.61237244
\end{align*}
\]

Only the first, third and fourth solutions satisfy \( 2x \leq y \) and \( 2z \leq y \). From all of these solutions the largest value of \( e \) is \( e = 0.42459536 \), so if a curved pyramid with this ratio is maintained, then \( p \geq \frac{201326679}{34359738368} \).

Now the exact value of this particular \( e \) is rounded up to a representable F32:

\[
e_3 = \frac{14247057}{2^{25}} \approx 0.4245953858
\]

The Singular code used to perform these operations is below.

```singular
ring r=0,(x,y,z,e),lp;
poly p = z6+x2y2*(x2+y2-3z2);
ideal I = x2+y2+z2-1,x-e;
poly q = diff(reduce(p,groebner(I)),z);
ideal J = p-201326679/34359738368,I,q;
LIB "solve.lib";
def R = solve(groebner(J));
setring R;
SOL[6];
SOL[8];
SOL[36];
SOL[40];
SOL[44];
```

340
ring r=0,(z,y,x,e),lp;
poly p = z6+x2y2*(x2+y2-3z2);
ideal I = x2+y2+z2-1,z-e;
poly q = diff(reduce(p,groebner(I)),x);
ideal J = p-201326679/34359738368,I,q;
LIB "solve.lib";
def R = solve(groebner(J));
setring R;
SOL[4];
SOL[24];
SOL[28];

ring r=0,(x,y,z,e),lp;
ideal I = z6+x2y2*(x2+y2-3z2)-201326679/34359738368,x2+y2+z2-1,x-e,z;
LIB "solve.lib";
def R = solve(groebner(I));
setring R;
SOL[6];
SOL[8];

ring r=0,(x,y,z,e),lp;
ideal I = z6+x2y2*(x2+y2-3z2)-201326679/34359738368,x2+y2+z2-1,x,z-e;
LIB "solve.lib";
def R = solve(groebner(I));
setring R;
SOL[4];

ring r=0,(x,y,z,e),lp;
ideal I = z6+x2y2*(x2+y2-3z2)-201326679/34359738368,x2+y2+z2-1,x-e,z-e;
LIB "solve.lib";
def R = solve(groebner(I));
setring R;
SOL[8];
SOL[10];
SOL[12];
D.2. The Conditions for the Use of F64

Repeating the process for F64, it is required to calculate, for \( p = z^6 + x^2y^2(x^2 + y^2 - 3z^2) \):

\[
\min_{e > 0} \left( \frac{6 \left( (1 + 2^{-52})^8 - 1 \right)}{\min \left( \left| \mathbf{x} \right| = 1, x, y, z \geq 0, 2x \leq y, 2z \leq y, \begin{array}{l} x^2 \geq e^2(x^2 + y^2 + z^2) \\
\quad \text{or} \\
\quad z^2 \geq e^2(x^2 + y^2 + z^2) \end{array} \right)} < 2^{-10} \right)
\]

Rearranging and simplifying:

\[
\min_{e > 0} \left( 3 \times 2^{11} \left( (1 + 2^{-52})^8 - 1 \right) \right) \leq \min \left( p(\mathbf{x}) : \begin{array}{l} \left| \mathbf{x} \right| = 1, x, y, z \geq 0, \\
2x \leq y, 2z \leq y, \\
(x \geq e \quad \text{or} \quad z \geq e) \end{array} \right)
\]

Taking the first two terms of the expansion of \((1 + 2^{-52})^8\) and rounding up will give a conservative estimate on \( e \):

\[
\min_{e > 0} \left( \frac{108086391056891991}{2^{93}} \right) \leq \min \left( p(\mathbf{x}) : \begin{array}{l} x^2 + y^2 + z^2 = 1, x, y, z \geq 0, \\
2x \leq y, 2z \leq y, \\
(x \geq e \quad \text{or} \quad z \geq e) \end{array} \right)
\]

Now the minimum of \( p \) will occur closest to the \( y \) axis, so first assuming \( x = e \):

\[
\min_{e > 0} \left( \frac{108086391056891991}{2^{93}} \right) \leq \min \left( p(\mathbf{x}) : \begin{array}{l} x^2 + y^2 + z^2 = 1, x, y, z \geq 0, \\
2x \leq y, 2z \leq y, \\
(x = e \quad \text{or} \quad z \geq e) \end{array} \right)
\]

Now consider the points which satisfy:

\[
p = \frac{108086391056891991}{2^{93}} \quad x^2 + y^2 + z^2 = 1 \quad x = e \quad \text{(D.5)}
\]

Substituting the latter conditions into \( p \) gives:

\[
p(z, e) = z^6 + 4z^4e^2 + 4z^2e^4 - 5z^2e^2 - e^4 + e^2
\]
The minimum of this expression will occur when $\frac{\partial p}{\partial z} = 0$, namely:

$$6z^5 + 16z^3e^2 + 8ze^4 - 10ze^2 = 0 \quad (D.6)$$

Together, the equations in D.5 and D.6, are four equations in four unknowns. These equations can be solved using Gröbner bases. The solutions which have $x, y, z \geq 0$ are:

- $x = 0.000033036247 \quad y = 1 \quad z = 0 \quad e = 0.000033036247$
- $x = 1 \quad y = 0.000033036247 \quad z = 0 \quad e = 1$
- $x = 0.57734522 \quad y = 0.57735496 \quad z = 0.57735063 \quad e = 0.57734522$
- $x = 0.57735532 \quad y = 0.57734558 \quad z = 0.57734991 \quad e = 0.57735532$

Only the first and third solutions satisfy $2x \leq y$ and $2z \leq y$. It can be confirmed that these points are both minima of $p$ by inspecting $\frac{\partial^2 p}{\partial z^2}$.

Repeating the exercise for $z = e$:

$$\min_{e > 0} \left( \frac{108086391056891991}{2^{93}} \right) \leq \left( p(x) : \begin{array}{l} x^2 + y^2 + z^2 = 1, x, y, z \geq 0, \\ 2x \leq y, 2z \leq y, \\ (x \geq e \text{ or } z = e) \end{array} \right)$$

Now consider the points which satisfy:

$$p = \frac{108086391056891991}{2^{93}} \quad x^2 + y^2 + z^2 = 1 \quad z = e \quad (D.7)$$

Substituting the latter conditions into $p$ gives:

$$p(x, e) = 4x^4e^2 - x^4 + 4x^2e^4 - 5x^2e^2 + x^2 + e^6$$

The minimum of this expression will occur when $\frac{\partial p}{\partial x} = 0$, namely:

$$16x^3e^2 - 4x^3 + 8xe^4 - 10xe^2 + 2x = 0 \quad (D.8)$$

Together, the equations in D.7 and D.8, are four equations in four unknowns. These equations can be solved using Gröbner bases. The variable $e$ must

343
The solutions which have $x, y, z \geq 0$ are:

\[
\begin{align*}
&x = 0 \quad y = 1 \quad z = 0 \quad e = 0 \\
&x = 0.57735122 \quad y = 0.57735122 \quad z = 0.57734836 \quad e = 0.57734836 \\
&x = 0.57734932 \quad y = 0.57734932 \quad z = 0.57735218 \quad e = 0.57735218
\end{align*}
\]

Only the first solution satisfies $2x \leq y$ and $2z \leq y$. It can be confirmed that this point is a minimum of $p$ by inspecting $\partial^2 p / \partial z^2$.

This has investigated the turning points on the curved pyramid. The corners of the curved pyramid would need to satisfy:

\[
p = \frac{108086391056891991}{2^{93}} x^2 + y^2 + z^2 = 1 \quad (x, z) = (0, e), (e, 0), (e, e)
\]

The solutions from these sets of equations are:

\[
\begin{align*}
&x = 0.0000033036247 \quad y = 1 \quad z = 0 \quad e = 0.0000033036247 \\
&x = 1 \quad y = 0.0000033036247 \quad z = 0 \quad e = 1 \\
&x = 0 \quad y = 1 \quad z = 0 \quad e = 0 \\
&x = 0.0000033036247 \quad y = 1 \quad z = 0.0000033036247 \quad e = 0.0000033036247 \\
&x = 0.57734862 \quad y = 0.57735357 \quad z = 0.57734862 \quad e = 0.57734862 \\
&x = 0.57735192 \quad y = 0.57734697 \quad z = 0.57735192 \quad e = 0.57735192
\end{align*}
\]

Only the first, third and fourth solutions satisfy $2x \leq y$ and $2z \leq y$. From all of these solutions the largest value of $e$ is $e = 0.0000033036482$, so if a curved pyramid with this ratio is maintained, then $p \geq \frac{108086391056891991}{2^{93}}$. Now the exact value of this particular $e$ is rounded up to a representable
F32:

\[ e_4 = \frac{14529599}{2^{42}} \approx 0.000003303648327 \]

The Singular code used to perform these operations is below.

```c
ring r=0,(x,y,z,e),lp;
poly p = z6+x2y2*(x2+y2-3z2);
ideal I = x2+y2+z2-1,x-e;
poly q = diff(reduce(p,groebner(I)),z);
ideal J = p-108086391056891991/990352031428304219993993792,I,q;
LIB "solve.lib";
def R = solve(groebner(J));
setring R;
SOL[6];
SOL[8];
SOL[36];
SOL[40];
SOL[44];

ring r=0,(z,y,x,e),lp;
poly p = z6+x2y2*(x2+y2-3z2);
ideal I = x2+y2+z2-1,z-e;
poly q = diff(reduce(p,groebner(I)),x);
ideal J = p-108086391056891991/99035203142830421999192993792,I,q;
LIB "solve.lib";
def R = solve(groebner(J));
setring R;
SOL[2];
SOL[4];
SOL[6];
SOL[8];
SOL[10];
SOL[12];
SOL[24];
SOL[28];

ring r=0,(x,y,z,e),lp;
```

345
ideal I = z6+x2y2*(x2+y2-3z2)
-108086391056891991/9903520314283042199192993792,
x2+y2+z2-1,x-e,z;
LIB "solve.lib";
def R = solve(groebner(I));
setring R;
SOL[6];
SOL[8];
ring r=0,(x,y,z,e),lp;
ideal I = z6+x2y2*(x2+y2-3z2)
-108086391056891991/9903520314283042199192993792,
x2+y2+z2-1,x,z-e;
LIB "solve.lib";
def R = solve(groebner(I));
setring R;
SOL[2];
ring r=0,(x,y,z,e),lp;
ideal I = z6+x2y2*(x2+y2-3z2)
-108086391056891991/9903520314283042199192993792,
x2+y2+z2-1,x-e,z-e;
LIB "solve.lib";
def R = solve(groebner(I));
setring R;
SOL[8];
SOL[10];
SOL[12];
E. The Condition for the Use of F32/F64 Far from the Variety

This appendix is associated with Chapter 7 and in particular the accurate evaluation of the Motzkin polynomial. Accurate evaluation far from the variety requires less precision and this appendix determines the region where F32 and F64 can be used.

E.1. The Conditions for the Use of F32

It is required to calculate, for \( p = z^6 + x^2 y^2 (x^2 + y^2 - 3z^2) \):

\[
\min_{e > 0} \left( \frac{6 \left( (1 + 2^{-23})^8 - 1 \right)}{6 \left( (1 + 2^{-23})^8 - 1 \right)} \right) < 2^{-10}
\]

\[
\min_{p(x)} \left( \begin{array}{c}
|\mathbf{x}| = 1, x, y, z \geq 0, \\
2z \leq y \leq 2x, x \leq y \quad \text{or} \quad 2z \leq x \leq 2y, y \leq x \quad \text{or} \\
2z \leq x \leq y, y \leq 2x \quad \text{or} \quad 2z \leq y \leq x, x \leq 2y \quad \text{or} \\
2x \leq y \leq 2z \quad \text{or} \quad 2y \leq x \leq 2z \\
(x^2 \geq e^2 (x^2 + y^2 + z^2)) \quad \text{or} \\
y^2 \geq e^2 (x^2 + y^2 + z^2) \quad \text{or} \\
z^2 \geq e^2 (x^2 + y^2 + z^2) \end{array} \right)
\]
Rearranging and simplifying:

\[
\min_{e>0} \left( 3 \times 2^{11} \left( (1 + 2^{-23})^8 - 1 \right) \right) < \min \left( p(x) : \begin{array}{l}
|\mathbf{x}| = 1, x, y, z \geq 0, \\
2z \leq y \leq 2x, x \leq y \quad \text{or} \quad 2z \leq x \leq 2y, y \leq x \\
2z \leq x \leq y, y \leq 2x \quad \text{or} \quad 2z \leq y \leq x, x \leq 2y \\
x \geq e \quad \text{or} \quad y \geq e \quad \text{or} \quad z \geq e
\end{array} \right) \]

Taking the first two terms of the expansion of \( (1 + 2^{-23})^8 \) and rounding up will give a conservative estimate on \( e \):

\[
\min_{e>0} \left( \frac{201326679}{34359738368} \right) \leq \min \left( p(x) : \begin{array}{l}
x^2 + y^2 + z^2 = 1, x, y, z \geq 0, \\
2z \leq y \leq 2x, x \leq y \quad \text{or} \quad 2z \leq x \leq 2y, y \leq x \\
2z \leq x \leq y, y \leq 2x \quad \text{or} \quad 2z \leq y \leq x, x \leq 2y \\
x \geq e \quad \text{or} \quad y \geq e \quad \text{or} \quad z \geq e
\end{array} \right) \]

Now the minimum of \( p \) will occur closest to the origin. So one of \( x, y \) or \( z \) will be \( e \). Given the symmetry in \( x \) and \( y \) consider only \( x = e \) and \( z = e \). so first assuming \( x = e \):

\[
\min_{e>0} \left( \frac{201326679}{34359738368} \right) \leq \min \left( p(x) : \begin{array}{l}
x^2 + y^2 + z^2 = 1, x, y, z \geq 0, \\
x \geq e \quad \text{or} \quad z \geq e
\end{array} \right)
\]

Now consider the points which satisfy:

\[
p = \frac{201326679}{34359738368} x^2 + y^2 + z^2 = 1 \quad x = e \quad \text{(E.1)}
\]

Substituting the latter conditions into \( p \) gives:

\[
p(z, e) = z^6 + 4z^4e^2 + 4z^2e^4 - 5ze^6 - e^4 + e^2
\]

The minimum of this expression will occur when \( \partial p/\partial z = 0 \), namely:

\[
6z^5 + 16z^3e^2 + 8ze^4 - 10ze^2 = 0 \quad \text{(E.2)}
\]
Together, the equations in E.1 and E.2, are four equations in four unknowns. These equations can be solved using Gröbner bases. The variable $e$ must satisfy:

\[
11682667931703362164193616740745216e^{14} - 29328364286880315433027725359579136e^{12} + 25576118536498892981215668421525504e^{10} - 8987880450068701936858640347561984e^8 + 1081419846482511806373812062126080e^6 - 18863893044946249761968854401024e^4 + 112807383575555458701226672128e^2 - 220327016256688764945910653 = 0
\]

The solutions which have $x, y, z \geq 0$ are:

- $x = 0.07677316, y = 0.99704859, z = 0, e = 0.07677316$
- $x = 0.99704859, y = 0.07677316, z = 0, e = 0.99704859$
- $x = 0.095825431, y = 0.93765632, z = 0.33409298, e = 0.095825431$
- $x = 0.44677538, y = 0.68891201, z = 0.57078192, e = 0.44677538$
- $x = 0.69570237, y = 0.45495471, z = 0.55589066, e = 0.69570237$

None of these satisfy the boundary conditions.

Repeating the exercise for $z = e$:

\[
\min_{e>0} \left( \frac{201326679}{34359738368} \right) \leq \left( p(x) : \begin{array}{c}
  x^2 + y^2 + z^2 = 1, x, y, z \geq 0, \\
  2x \leq y, 2z \leq y, \\
  (x \geq e \text{ or } z = e)
\end{array}\right)
\]

Now consider the points which satisfy:

\[
p = \frac{201326679}{34359738368} \quad x^2 + y^2 + z^2 = 1 \quad z = e \quad (E.3)
\]

Substituting the latter conditions into $p$ gives:

\[
p(x, e) = 4x^4e^2 - x^4 + 4x^2e^4 - 5x^2e^2 + x^2 + e^6
\]
The minimum of this expression will occur when $\partial p/\partial x = 0$, namely:

$$16x^3e^2 - 4x^3 + 8xe^4 - 10xe^2 + 2x = 0 \quad (E.4)$$

Together, the equations in E.3 and E.4, are four equations in four unknowns. These equations can be solved using Gröbner bases. The variable $e$ must satisfy:

$$2656331146614175432704e^{10} - 1770887431076116955136e^8$$
$$+ 288230373162414505984e^6 - 15564447038111219712e^4$$
$$+ 1037629802540749808e^2 - 1688850572557410927 = 0$$

The solutions which have $x, y, z \geq 0$ are:

$$x = 0 \quad y = 0.90538322 \quad z = 0.42459536 \quad e = 0.42459536$$
$$x = 0.59903995 \quad y = 0.59903995 \quad z = 0.53132126 \quad e = 0.53132126$$
$$x = 0.55481331 \quad y = 0.55481331 \quad z = 0.61997127 \quad e = 0.61997127$$

None of these satisfy the boundary conditions.

This has investigated the turning points within the domain. Now turning to the corners of the domain. The various boundaries of the domain are, assuming $x \leq y$ without loss of generality due to the symmetry in $x$ and $y$:

$$2z \leq y \leq 2x, x \leq y$$
$$2z \leq x \leq y, y \leq 2x$$
$$2x \leq y \leq 2z$$

The various points lying on the boundary are then:

$$x \leq y = 2z$$
$$2z \leq y = 2x$$
$$x/2 \leq y/2 \leq x = 2z$$
$$2z \leq x = y$$
$$2x = y \leq 2z$$
$$2x \leq y = 2z$$

350
Substituting \( x^2 + y^2 + z^2 = 1 \) produces the following \((x, y, z)\) points:

\[
\begin{align*}
(\sqrt{1-5z^2}, 2z, z) & \quad \sqrt{1-5z^2} \leq 2z \\
(x, 2x, \sqrt{1-5x^2}) & \quad \sqrt{1-5x^2} \leq x \\
(2z, \sqrt{1-5z^2}, z) & \quad z \leq \sqrt{1-5z^2}/2 \leq 2z \\
(y, y, \sqrt{1-2y^2}) & \quad 2\sqrt{1-2y^2} \leq y \\
(x, 2x, \sqrt{1-5x^2}) & \quad x \leq \sqrt{1-5x^2} \\
(\sqrt{1-5z^2}, 2z, z) & \quad \sqrt{1-5z^2} \leq z
\end{align*}
\]

Simplifying:

\[
\begin{align*}
(\sqrt{1-5z^2}, 2z, z) & \quad 1/3 \leq z \\
(x, 2x, \sqrt{1-5x^2}) & \quad x \leq 1/\sqrt{5} \\
(2z, \sqrt{1-5z^2}, z) & \quad 1/\sqrt{21} \leq z \leq 1/3 \\
(y, y, \sqrt{1-2y^2}) & \quad 2/3 \leq y \leq 1/\sqrt{2}
\end{align*}
\]

The intersection of these lines with the cube \( x, y, z = e \) and thus the points closest to the origin satisfying the conditions are the following points:

\[
\begin{align*}
(e, 2\sqrt{(1-e^2)/5}, \sqrt{(1-e^2)/5}) & \quad e \leq 2/3 \\
(\sqrt{1-5e^2/4}, e, e/2) & \quad 2/3 \leq e \\
(\sqrt{1-5e^2/2}, 2e, e) & \quad 1/3 \leq e \\
(e, 2e, \sqrt{1-5e^2}) & \quad e \leq 1/\sqrt{5} \\
(e/2, e, \sqrt{1-5e^2/4}) & \quad e \leq 2/\sqrt{5} \\
(\sqrt{(1-e^2)/5}, 2\sqrt{(1-e^2)/5}, e) & \\
(e, \sqrt{1-5e^2/4}, e/2) & \quad 2/\sqrt{21} \leq e \leq 2/3 \\
(2\sqrt{(1-e^2)/5}, e, \sqrt{(1-e^2)/5}) & \quad 2/3 \leq e \leq 4/\sqrt{21} \\
(2e, \sqrt{1-5e^2}, e) & \quad 1/\sqrt{21} \leq e \leq 1/3 \\
(e, e, \sqrt{1-2e^2}) & \quad 2/3 \leq e \leq 1/\sqrt{2} \\
(\sqrt{(1-e^2)/2}, \sqrt{(1-e^2)/2}, e) & \quad e \leq 1/3
\end{align*}
\]
The value of $p$ at these points is:

\[
\begin{align*}
-81e^6 + 63e^4 + 17e^2 + 1 & \quad e \leq 2/3 \\
\frac{81}{64} e^6 - \frac{9}{4} e^4 + e^2 & \quad 2/3 \leq e \\
81e^6 - 36e^4 + 4e^2 & \quad 1/3 \leq e \\
-45e^6 + 63e^4 - 15e^2 + 1 & \quad e \leq 1/\sqrt{5} \\
-\frac{45}{64} e^6 + \frac{63}{16} e^4 - \frac{15}{4} e^2 + 1 & \quad e \leq 2/\sqrt{5} \\
9e^6 + 36e^4 - 24e^2 + 4 & \quad \\
\frac{81}{64} e^6 - \frac{9}{4} e^4 + e^2 & \quad 2/\sqrt{21} \leq e \leq 2/3 \\
-81e^6 + 63e^4 + 17e^2 + 1 & \quad 2/3 \leq e \leq 4/\sqrt{21} \\
81e^6 - 36e^4 + 4e^2 & \quad 1/\sqrt{21} \leq e \leq 1/3 \\
9e^4 - 6e^2 + 1 & \quad 2/3 \leq e \leq 1/\sqrt{2} \\
\frac{9e^4 - 6e^2 + 1}{4} & \quad e \leq 1/3 \\
\end{align*}
\]

Simplifying:

\[
\begin{align*}
\frac{(9e^2 + 1)^2 (1-e^2)}{125} & \quad e \leq 4/\sqrt{21} \\
\frac{e^2 (9e^2 - 8)^2}{64} & \quad 2/\sqrt{21} \leq e \\
e^2 (9e^2 - 2)^2 & \quad 1/\sqrt{21} \leq e \\
-45e^6 + 63e^4 - 15e^2 + 1 & \quad e \leq 1/\sqrt{5} \\
-\frac{45}{64} e^6 + \frac{63}{16} e^4 - \frac{15}{4} e^2 + 1 & \quad e \leq 2/\sqrt{5} \\
9e^6 + 36e^4 - 24e^2 + 4 & \quad 2/3 \leq e \leq 1/\sqrt{2} \\
\frac{(3e^2 - 1)^2}{4} & \quad e \leq 1/3 \\
\end{align*}
\]

If these minima were to equal the desired $\frac{201328679}{34359738368}$, then the only potential
values of $e$ are:

\[
\frac{201326679}{34359738368} = \frac{e^2(9e^2 - 8)^2}{64} \leq 2/\sqrt{21} = e
\]

\[
e = 0.90191337
e = 0.97897477
\]

\[
\frac{201326679}{34359738368} = \frac{e^2(9e^2 - 2)^2}{64} \leq 1/\sqrt{21} = e
\]

\[
e = 0.45095668
e = 0.48948738
\]

Rounding the largest of these to an F32 gives:

\[
e_5 = \frac{2053059}{2^{31}} \approx 0.9789748192
\]

The Singular code used to perform these operations is below.

```
ring r=0,(x,y,z,e),lp;
poly p = z6+x2y2*(x2+y2-3z2);
ideal I = x2+y2+z2-1,x-e;
poly q = diff(reduce(p,groebner(I)),z);
ideal J = p-201326679/34359738368,I,q;
LIB "solve.lib";
def R = solve(groebner(J));
setring R;
SOL[4];
```

```
ring r=0,(z,y,x,e),lp;
poly p = z6+x2y2*(x2+y2-3z2);
ideal I = x2+y2+z2-1,z-e;
poly q = diff(reduce(p,groebner(I)),x);
ideal J = p-201326679/34359738368,I,q;
LIB "solve.lib";
def R = solve(groebner(J));
setring R;
SOL[4];
```

353
E.2. The Conditions for the Use of F64

Repeating the process for F64, it is required to calculate, for \( p = z^6 + x^2y^2(x^2 + y^2 - 3z^2) \):

\[
\begin{align*}
\min_{e > 0} \left( \frac{6 \left( (1 + 2^{-52})^8 - 1 \right)}{||x|| = 1, x, y, z \geq 0,} \right)
\begin{cases}
(2z \leq y \leq 2x, x \leq y & \text{or} & 2z \leq x \leq 2y, y \leq x & \text{or} \\
2z \leq x \leq y, y \leq 2x & \text{or} & 2z \leq y \leq x, x \leq 2y & \text{or} \\
(2x \geq e & \text{or} & y \geq e & \text{or} & z \geq e) & \text{or} & x \geq e & \text{or} & y \geq e & \text{or} & z \geq e & \text{or} & \end{cases}
\end{align*}
\]

Rearranging and simplifying:

\[
\begin{align*}
\min_{e > 0} \left( 3 \times 2^{11} \left( (1 + 2^{-52})^8 - 1 \right) \right) < \min (p(x) : \\
||x|| = 1, x, y, z \geq 0, \\
(2z \leq y \leq 2x, x \leq y & \text{or} & 2z \leq x \leq 2y, y \leq x & \text{or} \\
2z \leq x \leq y, y \leq 2x & \text{or} & 2z \leq y \leq x, x \leq 2y & \text{or} \\
2x \leq y \leq 2z & \text{or} & 2y \leq x \leq 2z & \text{or} & x \geq e & \text{or} & y \geq e & \text{or} & z \geq e & \text{or} & \end{cases}
\end{align*}
\]
Taking the first two terms of the expansion of \((1 + 2^{-52})^8\) and rounding up will give a conservative estimate on \(e\):

\[
\min_{e > 0} \left( \frac{108086391056891991}{2^{93}} \right) \leq \min (p(x) : \begin{aligned}
x^2 + y^2 + z^2 &= 1, x, y, z \geq 0, \\
(2z \leq y \leq 2x, x \leq y) &\quad \text{or} \quad (2z \leq x \leq 2y, y \leq x, x \leq 2y) &\quad \text{or} \\
2x \leq y \leq 2z &\quad \text{or} \quad 2y \leq x \leq 2z \\
(x \geq e &\quad \text{or} \quad y \geq e &\quad \text{or} \quad z \geq e) 
\end{aligned}
\)

Now the minimum of \(p\) will occur closest to the origin. So one of \(x\), \(y\) or \(z\) will be \(e\). Given the symmetry in \(x\) and \(y\) consider only \(x = e\) and \(z = e\). so first assuming \(x = e\):

\[
\min_{e > 0} \left( \frac{108086391056891991}{2^{93}} \right) \leq \left( p(x) : \begin{aligned}
x^2 + y^2 + z^2 &= 1, x, y, z \geq 0, \\
2x \leq y, 2z \leq y, &\quad \text{or} \quad (x \geq e &\quad \text{or} \quad z \geq e) 
\end{aligned}
\right)
\]

Now consider the points which satisfy:

\[
p = \frac{108086391056891991}{2^{93}} \quad x^2 + y^2 + z^2 = 1 \quad x = e \quad \text{(E.5)}
\]

Substituting the latter conditions into \(p\) gives:

\[
p(z, e) = z^6 + 4z^4e^2 + 4z^2e^4 - 5z^2e^2 - e^4 + e^2
\]

The minimum of this expression will occur when \(\partial p/\partial z = 0\), namely:

\[
6z^5 + 16z^3e^2 + 8ze^4 - 10ze^2 = 0 \quad \text{(E.6)}
\]

Together, the equations in E.5 and E.6, are four equations in four unknowns. These equations can be solved using Gröbner bases. The solutions which
have $x, y, z \geq 0$ are:

\[
\begin{align*}
\text{x} & = 0.0000033036247 & \text{y} & = 1 & \text{z} & = 0 & \text{e} & = 0.0000033036247 \\
\text{x} & = 1 & \text{y} & = 0.0000033036247 & \text{z} & = 0 & \text{e} & = 1 \\
\text{x} & = 0.0000033036482 & \text{y} & = 0.99999787 & \text{z} & = 0.0020651821 & \text{e} & = 0.0000033036482 \\
\text{x} & = 0.57734522 & \text{y} & = 0.57735496 & \text{z} & = 0.57735063 & \text{e} & = 0.57734522 \\
\text{x} & = 0.57735532 & \text{y} & = 0.57734558 & \text{z} & = 0.57734991 & \text{e} & = 0.57735532
\end{align*}
\]

None of these satisfy the boundary conditions.

Repeating the exercise for $z = e$:

\[
\min_{e > 0} \left( \frac{108086391056891991}{2^{93}} \right) \left( \begin{array}{c}
x^2 + y^2 + z^2 = 1, x, y, z \geq 0, \\
p(x) : \begin{array}{l}2x \leq y, 2z \leq y, \\
(x \geq e \text{ or } z = e)\end{array} \end{array} \right)
\]

Now consider the points which satisfy:

\[
p = \frac{108086391056891991}{2^{93}} x^2 + y^2 + z^2 = 1 \quad z = e \quad (E.7)
\]

Substituting the latter conditions into $p$ gives:

\[
p(x, e) = 4x^4e^2 - x^4 + 4x^2e^4 - 5x^2e^2 + x^2 + e^6
\]

The minimum of this expression will occur when $\partial p/\partial x = 0$, namely:

\[
16x^3e^2 - 4x^3 + 8xe^4 - 10xe^2 + 2x = 0 \quad (E.8)
\]

Together, the equations in E.7 and E.8, are four equations in four unknowns. These equations can be solved using Gröbner bases. The variable $e$ must
satisfy:

\[
22067935788468795603601971909644522441098433594438713344e^{10}
- 147119571923125330402401314606429681627398955729625808896e^8
+ 245199286527837859642040816620049651118188033272860114944e^6
- 240848048144130923789959094351533534674419712e^4
+ 1605653654294206158599972729567689023116279808e^2
- 267608942370685025168292073956722509879435887 = 0
\]

The solutions which have \(x, y, z \geq 0\) are:

\[
\begin{align*}
  x = 0 & \quad y = 1 \quad z = 0 \quad e = 0 \\
  x = 0.57734836 & \quad y = 0.57735122 \quad z = 0.57735122 \quad e = 0.57734836 \\
  x = 0.57735218 & \quad y = 0.57734932 \quad z = 0.57734932 \quad e = 0.57735218 \\
\end{align*}
\]

None of these satisfy the boundary conditions.

This has investigated the turning points within the domain. Now turning to the corners of the domain. The various boundaries of the domain are, assuming \(x \leq y\) without loss of generality due to the symmetry in \(x\) and \(y\):

\[
\begin{align*}
  2z & \leq y \leq 2x, \ x \leq y \\
  2z & \leq x \leq y, \ y \leq 2x \\
  2x & \leq y \leq 2z \\
\end{align*}
\]

The various points lying on the boundary are then:

\[
\begin{align*}
  x \leq y = 2z \\
  2z \leq y = 2x \\
  x/2 \leq y/2 \leq x = 2z \\
  2z \leq x = y \\
  2x = y \leq 2z \\
  2x \leq y = 2z \\
\end{align*}
\]
Substituting \( x^2 + y^2 + z^2 = 1 \) produces the following \((x, y, z)\) points:

\[
\begin{align*}
(\sqrt{1-5z^2}, 2z, z) & \quad \sqrt{1-5z^2} \leq 2z \\
(x, 2x, \sqrt{1-5x^2}) & \quad \sqrt{1-5x^2} \leq x \\
(2z, \sqrt{1-5z^2}, z) & \quad z \leq \sqrt{1-5z^2}/2 \leq 2z \\
(y, y, \sqrt{1-2y^2}) & \quad 2\sqrt{1-2y^2} \leq y \\
(x, 2x, \sqrt{1-5x^2}) & \quad x \leq \sqrt{1-5x^2} \\
(\sqrt{1-5z^2}, 2z, z) & \quad \sqrt{1-5z^2} \leq z
\end{align*}
\]

Simplifying:

\[
\begin{align*}
(\sqrt{1-5z^2}, 2z, z) & \quad 1/3 \leq z \\
(x, 2x, \sqrt{1-5x^2}) & \quad x \leq 1/\sqrt{5} \\
(2z, \sqrt{1-5z^2}, z) & \quad 1/\sqrt{21} \leq z \leq 1/3 \\
(y, y, \sqrt{1-2y^2}) & \quad 2/3 \leq y \leq 1/\sqrt{2}
\end{align*}
\]

The intersection of these lines with the cube \( x, y, z = e \) and thus the points closest to the origin satisfying the conditions are the following points:

\[
\begin{align*}
(e, 2\sqrt{(1-e^2)/5}, \sqrt{(1-e^2)/5}) & \quad e \leq 2/3 \\
(\sqrt{1-5e^2/4}, e, e/2) & \quad 2/3 \leq e \\
(\sqrt{1-5e^2}, 2e, e) & \quad 1/3 \leq e \\
(e, 2e, \sqrt{1-5e^2}) & \quad e \leq 1/\sqrt{5} \\
(e/2, e, \sqrt{1-5e^2/4}) & \quad e \leq 2/\sqrt{5} \\
(\sqrt{(1-e^2)/5}, 2\sqrt{(1-e^2)/5}, e) & \quad (e, \sqrt{1-5e^2/4}, e/2) \quad 2/\sqrt{21} \leq e \leq 2/3 \\
(2\sqrt{(1-e^2)/5}, e, \sqrt{(1-e^2)/5}) & \quad 2/3 \leq e \leq 4/\sqrt{21} \\
(2e, \sqrt{1-5e^2}, e) & \quad 1/\sqrt{21} \leq e \leq 1/3 \\
(e, e, \sqrt{1-2e^2}) & \quad 2/3 \leq e \leq 1/\sqrt{2} \\
(\sqrt{(1-e^2)/2}, \sqrt{(1-e^2)/2}, e) & \quad e \leq 1/3
\end{align*}
\]
The value of $p$ at these points is:

\[
\begin{align*}
-81e^6 + 63e^4 + 17e^2 + 1 & \quad e \leq 2/3 \\
\frac{81}{64}e^6 - \frac{9}{4}e^4 + e^2 & \quad 2/3 \leq e \\
\frac{81}{64}e^6 - 36e^4 + 4e^2 & \quad 1/3 \leq e \\
-45e^6 + 63e^4 - 15e^2 + 1 & \quad e \leq 1/\sqrt{5} \\
-\frac{45}{64}e^6 + \frac{63}{16}e^4 - \frac{15}{4}e^2 + 1 & \quad e \leq 2/\sqrt{5} \\
9e^6 + 36e^4 - 24e^2 + 4 & \\
\frac{81}{64}e^6 - \frac{9}{4}e^4 + e^2 & \quad 2/\sqrt{21} \leq e \leq 2/3 \\
\frac{-81e^6 + 63e^4 + 17e^2 + 1}{125} & \quad 2/3 \leq e \leq 4/\sqrt{21} \\
\frac{81e^6 - 36e^4 + 4e^2}{25} & \quad 1/\sqrt{21} \leq e \leq 1/3 \\
9e^4 - 6e^2 + 1 & \quad 2/3 \leq e \leq 1/\sqrt{2} \\
\frac{9e^4 - 6e^2 + 1}{4} & \quad e \leq 1/3 \\
\end{align*}
\]

Simplifying:

\[
\begin{align*}
\frac{(9e^2 + 1)^2(1 - e^2)}{125} & \quad e \leq 4/\sqrt{21} \\
\frac{e^2(9e^2 - 8)^2}{64} & \quad 2/\sqrt{21} \leq e \\
\frac{e^2(9e^2 - 2)^2}{64} & \quad 1/\sqrt{21} \leq e \\
-\frac{45e^6 + 63e^4 - 15e^2 + 1}{25} & \quad e \leq 1/\sqrt{5} \\
-\frac{45}{64}e^6 + \frac{63}{16}e^4 - \frac{15}{4}e^2 + 1 & \quad e \leq 2/\sqrt{5} \\
9e^6 + 36e^4 - 24e^2 + 4 & \\
\left(3e^2 - 1\right)^2 & \quad 2/3 \leq e \leq 1/\sqrt{2} \\
\frac{\left(3e^2 - 1\right)^2}{4} & \quad e \leq 1/3 \\
\end{align*}
\]

If these minima were to equal the desired $108086391056891991$ then the only
potential values of $e$ are:

\[
\frac{108086391056891991}{2^{93}} = \frac{(9e^2 + 1)^2(1 - e^2)}{125} \quad e \leq 4/\sqrt{21}
\]

\[
e = 0.33333268 \quad e = 0.33333986
\]

\[
\frac{108086391056891991}{2^{93}} = \frac{e^2(9e^2 - 8)^2}{64} \quad 2/\sqrt{21} \leq e
\]

\[
e = 0.94280739 \quad e = 0.94281069
\]

\[
\frac{108086391056891991}{2^{93}} = \frac{e^2(9e^2 - 2)^2}{1/\sqrt{21}} \leq e
\]

\[
e = 0.47140369 \quad e = 0.47140535
\]

Rounding the largest of these to an F32 gives:

\[
e_6 = \frac{15817739}{2^{34}} \approx 0.9428107142
\]

The Singular code used to perform these operations is below.

```
ring r=0,(x,y,z,e),lp;
poly p = z6+x2y2*(x2+y2-3z2);
ideal I = x2+y2+z2-1,x-e;
poly q = diff(reduce(p,groebner(I)),z);
ideal J = p-108086391056891991/9903520314283042199192993792,I,q;
LIB "solve.lib";
def R = solve(groebner(J));
setring R;
SOL[6];
SOL[8];
SOL[36];
SOL[40];
SOL[44];
```

```
ring r=0,(z,y,x,e),lp;
poly p = z6+x2y2*(x2+y2-3z2);
ideal I = x2+y2+z2-1,z-e;
poly q = diff(reduce(p,groebner(I)),x);
ideal J = p-108086391056891991/9903520314283042199192993792,I,q;
LIB "solve.lib";
```
def R = solve(groebner(J));
setring R;
SOL[4];
SOL[24];
SOL[28];

ring r=0,(e),lp;
ideal I = (108086391056891991/9903520314283042199192993792) - (1/125)*(9*e2-1)^2*(1-e2);
solve(groebner(I));
F. Calculation of ∆ and $p_{\text{min}}$ Near $x = y = z$

This appendix is associated with Chapter 7 and in particular the accurate evaluation of the Motzkin polynomial.

F.1. Calculation of ∆

Close to the variety $x = y = z$, the evaluation of $p$ requires the implementation of only the dominant terms of $p$, $p_{\text{dom}}$. It is required to know the maximum absolute relative error in using $p_{\text{dom}}$ to evaluate $p$ so it is required to calculate:

$$\Delta = \max \left| \frac{p_{\text{dom}} - p}{p} \right|$$

Where

$$p = z^6 + x^2y^2(x^2 + y^2 - 3z^2)$$
$$p_{\text{dom}} = 4z^4(x - z)^2 + 4z^4(x - z)(y - z) + 4z^4(y - z)^2$$

Over the domain defined by:

$$x \leq 2y \quad y \leq 2x \quad x \leq 2z \quad y \leq 2z$$
$$(x - z)^2 < e_2^2(x^2 + y^2 + z^2)$$
$$(y - z)^2 < e_2^2(x^2 + y^2 + z^2)$$

Given that both $p$ and $p_{\text{dom}}$ are homogeneous, the variables can be all be scaled, leaving $(p_{\text{dom}} - p)/p$ unchanged. Therefore assume $x^2 + y^2 + z^2 = 1$. The largest relative difference will occur furthest from $x = y = z$ and given the symmetry in $x$ and $y$ it can be assumed without loss of generality that
\[ x - z = e \text{ where } e = \pm e_2. \] Further, consider the function:

\[
q = p_{dom} - p - fp
\]
\[
= -8yz^5 + 4yz^4e - 9z^6 - 24z^5ef - 36z^5e - 28z^4e^2f - 28z^4e^2
+ 6z^4f + 10z^4 - 16z^3e^3f - 16z^3e^3 + 14z^3ef + 14z^3e - 4z^2e^4f - 4z^2e^4
+ 11z^2e^2f + 11z^2e^2 - z^2f - z^2 + 4ze^3f + 4ze^3 - 2zef - 2ze + e^4f + e^4
- e^2f - e^2
\]

Where \( y^2 + 2z^2 + 2ze + e^2 - 1 = 0. \) If there is a value of \( f \) such that \( q = 0 \) and \( dq/dz = 0 \) then \( q | f = \Delta, \) therefore it is required to solve:

\[
e = \pm e_2
\]
\[
y^2 + 2z^2 + 2ze + e^2 - 1 = 0
\]
\[
0 = -8yz^5 + 4yz^4e - 9z^6 - 24z^5ef - 36z^5e - 28z^4e^2f
- 28z^4e^2 + 6z^4f + 10z^4 - 16z^3e^3f - 16z^3e^3 + 14z^3ef + 14z^3e
- 4z^2e^4f - 4z^2e^4 + 11z^2e^2f + 11z^2e^2 - z^2f
- z^2 + 4ze^3f + 4ze^3 - 2zef - 2ze + e^4f + e^4 - e^2f - e^2
\]
\[
0 = y \frac{\partial q}{\partial z} + \frac{\partial q}{\partial y} \frac{dy}{dz}
\]
\[
= y \frac{\partial q}{\partial z} - (2z + e) \frac{\partial q}{\partial y}
\]
\[
= -40y^2z^4 + 16y^2z^3e - 54yz^5f - 78yz^5e - 120yz^4e - 180yz^4e
- 112yz^3ef - 112yz^3e2 + 24yz^3f + 40yz^3e^3f - 48yz^3e^3
+ 42yz^2ef + 42yz^2e - 8yze^4f - 8yze^4 + 22yze^2f + 22yze^2 - 2yzf - 2yz
+ 4yze^3f + 4yze^3 - 2yef - 2ye + 16z^6 - 4z^4e^2
\]

That is four equations in four unknowns but none of their solutions satisfies the domain restrictions. Therefore the corners of the domain will stress the relative size. Substituting \( x^2 + y^2 + z^2 = 1, \) \( x - z = \pm e_2 \) and \( y - z = \pm e_2 \)

\[ 363 \]
in \((p_{dom} - p)/p\) gives the following functions:

\[
\begin{align*}
&\frac{e_2(84z - 31e_2 + 64ze_2^2 + 80e_2^3)}{3(-4 + 4ze_2 + 5e_2^2)} \quad \text{where } z = \frac{\sqrt{3 - 2e_2^2} - 2e_2}{3} \\
&\frac{e_2(84z + 31e_2 + 64ze_2^2 - 80e_2^3)}{3(-4 - 4ze_2 + 5e_2^2)} \quad \text{where } z = \frac{\sqrt{3 - 2e_2^2} + 2e_2}{3} \\
&\frac{3e_2^2(5 - 16e_2^2)}{64e_2^2 - 31e_2^2 + 4}
\end{align*}
\]

The largest of these is the second, which rounded up to an F32 is:

\[
\Delta = \frac{6356645}{2^{39}} \approx 0.00001156266990
\]

The Singular code used to perform these operations is:

```singular
ring r=0,(x,y,z,e),lp;
poly p = z6+x2y2*(x2+y2-3z2);
poly pd = 4z4*(x-z)^2 + 4z4*(x-z)*(y-z) + 4z4*(y-z)^2;
ideal I = x2+y2+z2-1;
reduce(pd-p,groebner(I));
reduce(p,groebner(I));

ring r=0,(x,y,z,e,f),lp;
poly p = z6+x2y2*(x2+y2-3z2);
poly pd = 4z4*(x-z)^2 + 4z4*(x-z)*(y-z) + 4z4*(y-z)^2;
ideal I = x2+y2+z2-1,x-z-e;
poly q = reduce(pd-p-f*p,groebner(I));
poly qq = y*diff(q,z)-(2z+e)*diff(q,y);
ideal J = I,q,qq,e-12582933/2199023255552;
LIB "solve.lib";
def R = solve(groebner(J));
setring R;
SOL[18];

ring r=0,(x,y,z,e,f),lp;
poly p = z6+x2y2*(x2+y2-3z2);
poly pd = 4z4*(x-z)^2 + 4z4*(x-z)*(y-z) + 4z4*(y-z)^2;
ideal I = x2+y2+z2-1,x-z-e;
```

364
poly q = reduce(pd-p-f*p,groebner(I));
poly qq = y*diff(q,z)-(2z+e)*diff(q,y);
ideal J = I,q,qq,e+12582933/2199023255552;
LIB "solve.lib";
def R = solve(groebner(J));
setring R;
SOL[18];

ring r=0,(x,y,z,e,f),lp;
poly p = z6+x2y2*(x2+y2-3z2);
poly q = 4z4*(x-z)^2 + 4z4*(x-z)*(y-z) + 4z4*(y-z)^2 - p;
ideal I = x2+y2+z2-1,y-z-e,x-z-e;
ideal J = groebner(I);
reduce(q,J);
reduce(p,J);

ring r=0,(x,y,z,e,f),lp;
poly p = z6+x2y2*(x2+y2-3z2);
poly q = 4z4*(x-z)^2 + 4z4*(x-z)*(y-z) + 4z4*(y-z)^2 - p;
ideal I = x2+y2+z2-1,y-z-e,x-z+e;
ideal J = groebner(I);
reduce(q,J);
reduce(p,J);

F.2. Calculation of $p_{\min}$

Accurate evaluation close to the variety $x = y = z$ requires knowing the smallest value of $p$ just off the variety. Points $(x, y, z)$ which are F16 numbers, that are just off the variety $x = y = z$ are of the form (note that the
symmetry of $x$ and $y$ has been used to simplify the number of such points):

$$(2^e, 2^e, 2^e(1 - 2^{-11}))$$

$$(2^e(1 - 2^{-11}), 2^e, 2^e)$$

$$(2^e(1 + (m - 1)2^{-10}), 2^e(1 + m2^{-10}), 2^e(1 + m2^{-10})) \quad 0 < m < 2^{10}$$

$$(2^e(1 + m2^{-10}), 2^e(1 + m2^{-10}), 2^e(1 + (m - 1)2^{-10})) \quad 0 < m < 2^{10}$$

$$(2^e(1 + (m + 1)2^{-10}), 2^e(1 + m2^{-10}), 2^e(1 + m2^{-10})) \quad 0 \leq m < 2^{10} - 1$$

$$(2^e(1 + m2^{-10}), 2^e(1 + m2^{-10}), 2^e(1 + (m + 1)2^{-10})) \quad 0 \leq m < 2^{10} - 1$$

where $-14 \leq e \leq 15$

The calculation of $p_{\text{min}}$ requires all of the these numbers to satisfy $||x|| = 1$, normalising results:

$$\frac{(1, 1, 1 - 2^{-11})}{\sqrt{3 - 2^{-10} + 2^{-22}}}$$

$$\frac{(1 - 2^{-11}, 1, 1)}{\sqrt{3 - 2^{-10} + 2^{-22}}}$$

$$\frac{(1 + (m - 1)2^{-10}, 1 + m2^{-10}, 1 + m2^{-10})}{\sqrt{3 + (3m - 1)2^{-9} + (3m^2 - 2m + 1)2^{-20}}}$$

$$\frac{(1 + m2^{-10}, 1 + m2^{-10}, 1 + (m - 1)2^{-10})}{\sqrt{3 + (3m - 1)2^{-9} + (3m^2 - 2m + 1)2^{-20}}}$$

$$\frac{(1 + (m + 1)2^{-10}, 1 + m2^{-10}, 1 + (m - 1)2^{-10})}{\sqrt{3 + (3m + 1)2^{-9} + (3m^2 + 2m + 1)2^{-20}}}$$

$$\frac{(1 + m2^{-10}, 1 + m2^{-10}, 1 + (m + 1)2^{-10})}{\sqrt{3 + (3m + 1)2^{-9} + (3m^2 + 2m + 1)2^{-20}}}$$

$$0 < m < 2^{10}$$

$$0 \leq m < 2^{10} - 1$$
Substituting into \( p_{\text{dom}} \) gives:

\[
\frac{3 \times 2^{-20} (1 - 2^{-11})^4}{(3 - 2^{-10} + 2^{-22})^3} \quad 0 < m < 2^{10}
\]

\[
\frac{2^{-18} (1 + m 2^{-10})^4}{(3 + (3m - 1) 2^{-9} + (3m^2 - 2m + 1) 2^{-20})^3} \quad 0 < m < 2^{10} - 1
\]

Discarding obviously larger values:

\[
\frac{2^{-20}}{(3 - 2^{-10} + 2^{-22})^3} \quad 0 < m < 2^{10} - 1
\]

The latter is smaller if:

\[
474989023199232 m^6 + 2919282536582479872 * m^5 + 5485492432906190073855 * m^4
\]

\[
+ 2057992558161140247425024 * m^3 - 467786289637946511349231616 * m^2
\]

\[
- 533432231084826911576373696216 * m - 1639666436066962189356758948904960 > 0
\]

\( 0 \leq m < 2^{10} - 1 \)

However the only positive root of this polynomial is 1022, so the minimum is:

\[
\frac{2^{-20}}{(3 - 2^{-10} + 2^{-22})^3} = \frac{70368744177664}{199030912851459264513} \approx 3.535577844 \times 10^{-8}
\]
G. Calculation of $\Delta$ and $p_{min}$ Near $x = z = 0$

This appendix is associated with Chapter 7 and in particular the accurate evaluation of the Motzkin polynomial.

G.1. Calculation of $\Delta$

Close to the variety $x = z = 0$, the evaluation of $p$ requires the implementation of only the dominant terms of $p$, $p_{dom}$. It is required to know the maximum absolute relative error in using $p_{dom}$ to evaluate $p$ so it is required to calculate:

$$\Delta = \max \left| \frac{p_{dom} - p}{p} \right|$$

Where

$$p = z^6 + x^2y^2(x^2 + y^2 - 3z^2)$$

$$p_{dom} = z^6 + x^2y^4$$

Over the domain defined by:

$$2x \leq y \quad 2z \leq y$$

$$x^2 < e_1(x^2 + y^2 + z^2)$$

$$z^2 < e_2(x^2 + y^2 + z^2)$$

Given that both $p$ and $p_{dom}$ are homogeneous, the variables can be all be scaled, leaving $(p_{dom} - p)/p$ unchanged. Therefore assume $x^2 + y^2 + z^2 = 1$. The largest relative difference will occur furthest from $z = x = 0$ so first
assuming that $z = e$ where $e = \pm e_4$. Further, consider the function:

$$q = p_{dom} - p - fp$$

$$= -y^6 - 4y^4e^2f - 5y^4e^2 + y^4f + 2y^4 - 4y^2e^4f - 4y^2e^4$$

$$+ 5y^2e^2f + 5y^2e^2 - y^2f - y^2 - e^6f$$

If there is a value of $f$ such that $q = 0$ and $dq/dy = 0$ then $q \mid f = \Delta$, therefore it is required to solve:

$$e = \pm e_4$$

$$0 = -y^6 - 4y^4e^2f - 5y^4e^2 + y^4f + 2y^4 - 4y^2e^4f - 4y^2e^4$$

$$+ 5y^2e^2f + 5y^2e^2 - y^2f - y^2 - e^6f$$

$$0 = -6y^5 - 16y^3e^2f - 20y^3e^2 + 4y^3f + 8y^3 - 8ye^4f - 8ye^4$$

$$+ 10ye^2f + 10ye^2 - 2yf - 2y$$

That is three equations in three unknowns but none of their solutions satisfies the domain restrictions.

Now looking into $x = e$:

$$q = p_{dom} - p - fp$$

$$= -z^6f - 4z^4e^2f - 3z^4e^2 - 4z^2e^4f - 2z^2e^4 + 5z^2e^2f$$

$$+ 3z^2e^2 + e^6 + e^4f - e^4 - e^2f$$

If there is a value of $f$ such that $q = 0$ and $dq/dy = 0$ then $q \mid f = \Delta$, therefore it is required to solve:

$$e = \pm e_4$$

$$0 = -z^6f - 4z^4e^2f - 3z^4e^2 - 4z^2e^4f - 2z^2e^4 + 5z^2e^2f$$

$$+ 3z^2e^2 + e^6 + e^4f - e^4 - e^2f$$

$$0 = -6z^5f - 16z^3e^2f - 12z^3e^2 - 8ze^4f - 4ze^4 + 10ze^2f + 6ze^2$$

That is three equations in three unknowns, the solutions that match the
domain are:

\[ x = 0.0000033036483 \quad y = 1 \quad z = 0 \quad f = -0.10914092e - 10 \]
\[ x = 0.0000033036483 \quad y = 0.99991198 \quad z = 0.013268052 \quad f = 0.00035225809 \]

Now turning to the corners of the domain will stress the relative size. Substituting \( x^2 + y^2 + z^2 = 1, \ z = \pm e_4 \) and \( x = \pm e_4 \) in \( (p_{dom} - p)/p \) gives the following functions:

\[
\frac{2e_4^2(1 - 2e_4^2)}{(3e_4^2 - 1)^2}
\]

The largest of these is the second of the previous solutions, which rounded up to an F32 is:

\[
\Delta = \frac{1512937}{2^{32}} \approx 0.0003522580955
\]

The Singular code used to perform these operations is:

```singular
ring r=0,(x,y,z,e,f),lp;
poly p = z6+x2y2*(x2+y2-3z2);
poly pd = z6+x2y4;
ideal I = x2+y2+z2-1,z-e;
poly q = reduce(pd-p-f*p,groebner(I));
poly qq = diff(q,y);
ideal J = I,q,qq,e-14529599/4398046511104;
LIB "solve.lib";
def R = solve(groebner(J));
setring R;
```

```singular
ring r=0,(x,y,z,e,f),lp;
poly p = z6+x2y2*(x2+y2-3z2);
poly pd = z6+x2y4;
ideal I = x2+y2+z2-1,z-e;
poly q = reduce(pd-p-f*p,groebner(I));
poly qq = diff(q,y);
ideal J = I,q,qq,e+14529599/4398046511104;
LIB "solve.lib";
```

370
def R = solve(groebner(J));
setring R;
SOL[18];

ring r=0,(x,y,z,e,f),lp;
poly p = z6+x2y2*(x2+y2-3z2);
poly pd = z6+x2y4;
ideal I = x2+y2+z2-1,x-e;
poly q = reduce(pd-p-f*p,groebner(I));
poly qq = diff(q,z);
ideal J = I,q,qq,e-14529599/4398046511104;
LIB "solve.lib";
def R = solve(groebner(J));
setring R;
SOL[2];
SOL[10];

ring r=0,(x,y,z,e,f),lp;
poly p = z6+x2y2*(x2+y2-3z2);
poly pd = z6+x2y4;
ideal I = x2+y2+z2-1,x-e;
poly q = reduce(pd-p-f*p,groebner(I));
poly qq = diff(q,z);
ideal J = I,q,qq,e+14529599/4398046511104;
LIB "solve.lib";
def R = solve(groebner(J));
setring R;

ring r=0,(x,y,z,e,f),lp;
poly p = z6+x2y2*(x2+y2-3z2);
poly q = z6+x2y4-p;
ideal I = x2+y2+z2-1,z-e,x-e;
ideal J = groebner(I);
reduce(q,J);
reduce(p,J);
ring r=0,(x,y,z,e,f),lp;
poly p = z6+x2y2*(x2+y2-3z2);
poly q = z6+x2y4-p;
ideal I = x2+y2+z2-1,z-e,x+e;
ideal J = groebner(I);
reduce(q,J);
reduce(p,J);

G.2. Calculation of $p_{\min}$

Accurate evaluation close to the variety $x = z = 0$ requires knowing the smallest value of $p$ just off the variety. Points $(x, y, z)$ which are F16 numbers, that are just off the variety $z = x = 0$ are of the form (note that the domain requires $2x \leq y$ and $2z \leq y$ and the inputs are bound by the F16 format):

$$(2^{-14}, y, 0) \quad 2^{-13} \leq y \leq 2^{15}(2 - 2^{-10})$$
$$(0, y, 2^{-14}) \quad 2^{-13} \leq y \leq 2^{15}(2 - 2^{-10})$$

The calculation of $p_{\min}$ requires all of these numbers to satisfy $||x|| = 1$, normalising results:

$$\frac{(2^{-14}, y, 0)}{\sqrt{2^{-28} + y^2}}$$
$$\frac{(0, y, 2^{-14})}{\sqrt{2^{-28} + y^2}}$$

Substituting into $p_{\text{dom}}$ gives:

$$\frac{2^{-28}y^4}{(2^{-28} + y^2)^3} \quad 2^{-13} \leq y \leq 2^{15}(2 - 2^{-10})$$
$$\frac{2^{-84}}{(2^{-28} + y^2)^3} \quad 2^{-13} \leq y \leq 2^{15}(2 - 2^{-10})$$
The minima of these is then:

\[
\begin{array}{c}
16 \\
25 \\
1326633748212654339654679834044399616 \\
1528011284900335516792540513584566618515073508491395073 \\
72057594037927936 \\
1528011284900335516792540513584566618515073508491395073
\end{array}
\]

The last value is the smallest, rounded up to an F32:

\[
\frac{8413227}{2^{147}} \approx 4.715776830 \times 10^{-38}
\]
H. Calculation of $p_{\text{min}}$ Just Off the Origin and Far from the Variety

This appendix is associated with Chapter 7 and in particular the accurate evaluation of the Motzkin polynomial. Accurate evaluation close to the origin requires knowing the smallest value of $p$ just off the origin and not close to the other subvarieties. It is required to calculate, for $p = z^6 + x^2y^2(x^2 + y^2 - 3z^2)$:

$$
\min_{p(x)} \begin{cases}
||x|| = 1, x, y, z \geq 0, \\
(2z \leq y \leq 2x, x \leq y) \text{ or } (2z \leq x \leq 2y, y \leq x) \text{ or } \\
(2z \leq x \leq y, y \leq 2x) \text{ or } (2z \leq y \leq x, x \leq 2y) \text{ or }
\end{cases}
$$

Simplifying:

$$
\min_{p(x)} \begin{cases}
x^2 + y^2 + z^2 = 1, x, y, z \geq 0, \\
(2z \leq y \leq 2x, x \leq y) \text{ or } (2z \leq x \leq 2y, y \leq x) \text{ or } \\
(2z \leq x \leq y, y \leq 2x) \text{ or } (2z \leq y \leq x, x \leq 2y) \text{ or } \\
2x \leq y \leq 2z \text{ or } 2y \leq x \leq 2z \\
x < e_6, y < e_6, z < e_6
\end{cases}
$$

Substituting $x^2 + y^2 + z^2 = 1$ into $p$ gives:

$$p = 4y^4z^2 - y^4 + 4y^2z^4 - 5y^2z^2 + y^2 + z^6$$
The turning points of the $p$ then satisfy $\partial p/\partial y = \partial p/\partial z = 0$. The turning points satisfy:

$$
12z^9 - 7z^7 + z^5 = 0 \\
12yz^5 - 7yz^3 + yz = 0 \\
4y^2z^3 - y^2z - 12z^7 + 3z^5 = 0 \\
8y^3z^2 - 2y^3 + 4yz^4 - 5yz^2 + y = 0 \\
2y^4 - 2y^2z^2 - y^2 + 36z^8 - 3z^6 = 0
$$

The first equation is satisfied if $z = 0, 3, 4$, only $z = 0$ lives inside the domain, simplifying in light of this:

$$
y - 2y^3 = 0
$$

So either $y = 0, 1/\sqrt{2}$. So the set of potential turning points $(x, y, z)$ are:

$$
(1, 0, 0) \quad (1/\sqrt{2}, 1/\sqrt{2}, 0)
$$

Only the latter point is on the domain of interest and returns a value of $p$ of $1/4$.

This has investigated the turning points within the domain. Now turning to the corners of the domain. The various boundaries of the domain are, assuming $x \leq y$ without loss of generality due to the symmetry in $x$ and $y$:

$$
2z \leq y \leq 2x, x \leq y \\
2z \leq x \leq y, y \leq 2x \\
2x \leq y \leq 2z
$$
The various points lying on the boundary are then:

\[ x \leq y = 2z \]
\[ 2z \leq y = 2x \]
\[ x/2 \leq y/2 \leq x = 2z \]
\[ 2z \leq x = y \]
\[ 2x = y \leq 2z \]
\[ 2x \leq y = 2z \]

Substituting \( x^2 + y^2 + z^2 = 1 \) produces the following \((x, y, z)\) points:

\[(\sqrt{1-5z^2}, 2z, z) \quad \sqrt{1-5z^2} \leq 2z\]
\[(x, 2x, \sqrt{1-5x^2}) \quad \sqrt{1-5x^2} \leq x\]
\[(2z, \sqrt{1-5z^2}, z) \quad z \leq \sqrt{1-5z^2}/2 \leq 2z\]
\[(y, y, \sqrt{1-2y^2}) \quad 2\sqrt{1-2y^2} \leq y\]
\[(x, 2x, \sqrt{1-5x^2}) \quad x \leq \sqrt{1-5x^2}\]
\[(\sqrt{1-5z^2}, 2z, z) \quad \sqrt{1-5z^2} \leq z\]

Simplifying and applying the \( e_6 \) constraint:

\[(x, 2x, \sqrt{1-5x^2}) \quad \sqrt{(1-e_6^2)/5} \leq x \leq 1/\sqrt{5}\]
\[(2z, \sqrt{1-5z^2}, z) \quad 1/\sqrt{21} \leq z \leq 1/3\]
\[(y, y, \sqrt{1-2y^2}) \quad 2/3 \leq y \leq 1/\sqrt{2}\]

Inserting these values into \( p \):

\[-45x^6 + 63x^4 - 15x^2 + 1 \quad \sqrt{(1-e_6^2)/5} \leq x \leq 1/\sqrt{5}\]
\[81z^6 - 36z^4 + 4z^2 \quad 1/\sqrt{21} \leq z \leq 1/3\] \hspace{1cm} (H.1)
\[9y^4 - 6y^2 + 1 \quad 2/3 \leq y \leq 1/\sqrt{2}\] \hspace{1cm} (H.2)

The first curve has turning points at:

\[-270x^5 + 252x^3 - 30x = 0\]

The turning point, which can be shown to be a minimum, that falls within
the valid range of $x$ is:

$$x = \sqrt{\frac{2}{5}} - \frac{1}{\sqrt{15}}$$

The value of $p$ at this point is:

$$p = \frac{236}{75} - \frac{32\sqrt{6}}{25} \approx 0.011319796$$

(H.3)

Now turning to equation H.1, the minimum occurs at the lowest end of the range $z = 1/\sqrt{21}$, substituting gives:

$$p = \frac{121}{1029} \approx 0.1175898931$$

Finally turning to equation H.2, the minimum occurs at the lowest end of the range $y = 2/3$, substituting gives:

$$p = \frac{1}{9} \approx 0.1111111111$$

Conclude that $p_{\text{min}} = \frac{236}{75} - \frac{32\sqrt{6}}{25}$.

The Singular code used to perform these operations is below.

```singular
ring r=0,(x,y,z,e),lp;
poly p = z6+x2y2*(x2+y2-3z2);
ideal I = x2+y2+z2-1;
poly q = reduce(p,groebner(I)); q;
ideal J = jacob(q);
LIB "solve.lib";
def R = solve(groebner(J));
setring R;

ring r=0,(z,y,x),lp;
poly p = z6+x2y2*(x2+y2-3z2);
ideal I = y-2x,z2-1+5x2;
reduce(p,groebner(I));

ring r=0,(x,y,z),lp;
```

377
poly p = z6+x2y2*(x2+y2-3z2);
ideal I = x-2z,y2-1+5z2;
reduce(p,groebner(I));

ring r=0,(x,z,y),lp;
poly p = z6+x2y2*(x2+y2-3z2);
ideal I = x-y,z2-1+2y2;
reduce(p,groebner(I));