

Florent de Dinechin
Martin Kumm

Application-Specific Arithmetic

Computing Just Right
for the Reconfigurable Computer
and the Dark Silicon Era



Springer

Application-Specific Arithmetic

Florent de Dinechin • Martin Kumm

Application-Specific Arithmetic

Computing Just Right
for the Reconfigurable Computer
and the Dark Silicon Era



Springer

Florent de Dinechin
CITI laboratory, INSA-Lyon
Villeurbanne, France

Martin Kumm
Fulda University of Applied Sciences
Fulda, Germany

ISBN 978-3-031-42807-4 ISBN 978-3-031-42808-1 (eBook)
<https://doi.org/10.1007/978-3-031-42808-1>

© Springer Nature Switzerland AG 2024

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Paper in this product is recyclable.

Preface

Computer arithmetic is the art of representing and manipulating numbers in a computer. This book is about *hardware* arithmetic, the art of building those elementary computing operators that sit at the core of a computer, and give it its name. A programmer who uses the symbols + or / in a program probably knows that it will trigger hardware arithmetic operators, but many more operators are hidden from view. Talking through a mobile phone, paying with a credit card, driving a car, looking at a digital watch, all these activities rely on myriads of hardware arithmetic operators embedded in the artifacts of the digital age.

It is therefore not a surprise that computer arithmetic is among the first subjects taught in electrical engineering or computer science. Students learn very early how to build a binary adder, sometimes right after the basic gates (NOT, AND, OR, and XOR) have been introduced. Most students also quickly forget it, and only a select few will proceed to discover the more arcane aspects of hardware arithmetic, for instance the design of dividers, square root units, or trigonometric operators. This book is for them.

There are already many good books about computer arithmetic, and there are excellent courses in many universities. They perfectly cover the hardware implementation of $+$, $-$, \times , $/$, $\sqrt{}$, the five basic operations that have made the microprocessor so universal and so versatile in the twentieth century. This is mainstream hardware arithmetic, and these operators are here to stay. However, they are increasingly being complemented by a wild proliferation of new operators, diverse in shape and size, but sharing one common characteristic: they no longer need to be versatile, they no longer need to be universal, they need to do one thing for one application, and do it well (faster, or cheaper, or more accurately, or with less energy than the mainstream arithmetic operators). Multipliers that only multiply by a constant such as $\sin(\frac{17\pi}{1024})$ or $\log(2)$; fancy mathematical functions such as sigmoid, inverse cumulative distributions, or softmax; operators for low-precision data that would be catastrophic for general-purpose computing but seem perfectly suited to signal processing or machine learning; these are but a few examples of what researchers and application-specific circuit designers have been exploring in the past two decades. This book is about these application-specific operators.

As each application comes with its specific computing needs, there is an infinite number of application-specific operators. We therefore had to make a selection, and we apologize in advance to our readers if they do not find in these pages exactly the operator they need. However, the purpose of this book is not only to describe operators: it is also to describe methods, techniques that can be used to build all the operators that we do not describe. These methods were not developed for this book: they are the fruit of more than a decade of experiments and of development by the authors in the FloPoCo software project. FloPoCo is an open-source program that generates application-specific operators. Its operators have, for the most part, been published in various articles. The core techniques have not, or only in a rudimentary way. This is the main reason why we wanted to write this textbook. It fills a gap between existing textbooks and the state of the art.

The twenty-first century has also seen the maturation of the field programmable gate array (FPGA), and the emergence of reconfigurable computing which complements traditional microprocessors with FPGA-based accelerators. Some companies are specialized in this kind of acceleration, and some application domains are increasingly relying on it. When deploying an application on an FPGA, only the operators needed by this application need to be included: FPGA arithmetics, by definition, should be application-specific. This book is therefore especially relevant to FPGA-based computing, and it includes many FPGA-specific arithmetic techniques.

We hope that this book will be useful to students, to anybody interested in the design of computing circuits, to practitioners in VLSI design, and to engineers and researchers working with FPGAs. We hope it will save them some frustrating tinkering, and guide them on the path to the high-quality application-specific operators their projects deserve. We hope that they will find this book useful to their application domain, whatever it may be.¹ And we look forward to their feedback.

Lyon, France
Fulda, Germany

Florent de Dinechin
Martin Kumm

¹ ... with one huge exception: for lack of time we had to leave finite-field arithmetic and cryptography out of this book, and we deeply regret it.

Acknowledgments

Many people have contributed to making this book possible. Some have been our mentors, some have been our colleagues, some have been our students. Some began as students, then became colleagues. Many have become close friends. For these reasons and for many others, the authors would like to thank:

Levent Aksoy, Andrea Bocco, David Boland, Sylvie Boldo, Marthe Bonamy, Andreas Böttcher, Philip Brisk, Javier Bruguera, Nicolas Brunie, Chip-Hong Chang, Peter Cheung, Ray Cheung, Sylvain Chevillard, Maxime Christ, Caroline Collange, Marius Cornea, Octavian Cret̄, Marc Daumas, David Defour, Steven Derrien, Oregane Desrentes, Jérémie Detrey, Laurent-Stéphane Didier, Benoît Dupont de Dinechin, Yves Durand, Miloš Ercegovac, Alexey Ershov, Diana Fanghänel, Julian Faraone, Mathias Faust, Fabrizio Ferrandi, Nicolai Fiege, Silviu Filip, Luc Forget, Giulio Gambardella, Rémi Garcia, Mario Garrido, Alexandre Goldsztejn, Bernard Goossens, Jean-Marie Gorce, Oscar Gustafsson, Tobias Habermann, Martin Hardieck, Matei Istoan, Paolo Ienne, Claude-Pierre Jeannerod, Håkan Johansson, Mioara Joldeş, Petter Källström, Johannes Kappauf, Nachiket Kapre, Christian Klein, Marco Kleinlein, Harald Klingbeil, Andreas Koch, Dirk Koch, Jonas Kühle, Akash Kumar, Martin Langhammer, Philippe Langlois, Christoph Lauter, Vincent Lefèvre, Philip H. W. Leong, Nicolas Louvet, Wayne Luk, David Lutz, Sergey Maidanov, Peter Markstein, Steve McKeeever, Michael Mecik, Guillaume Melquiond, Uwe Meyer-Baese, Marc Mezzarobba, Konrad Möller, Lionel Morel, Duncan Moss, Jean-Michel Muller, Ettore Napoli, Andrey Naraikin, Stuart Oberman, Julian Oppermann, Bogdan Pasca, Jakoba Petri-König, Thomas Preußer, Patrice Quinton, Sanjay Rajopadhye, Melanie Reuter-Oppermann, Nathalie Revol, Guillaume Salagnac, Shahab Sanjari, Tapio Saramäki, Kentaro Sano, Olivier Sentieys, Nabeel Shiriazi, Patrick Sittel, Hayden So, Christine Solnon, Lukas Sommer, Antonio Strollo, Ping Tak Peter Tang, David Thomas, Arnaud Tisserand, Stephen Tridgell, Yohann Uguen, Álvaro Vázquez, Gilles Villard, Anastasia Volkova, Lukas Weber, Markus Weinhardt, John Wickerson, Paul Zimmermann, Peter Zipf, and many others.

It is a bit frightening to have found recollections of drinking beer while talking science with so many people.

The authors also want to express their gratitude to all the FloPoCo developers (with extra apologies to those whose work had to be left out of the present book):

Hatam Abdoli, Sebastian Banescu, Louis Besème, Andreas Böttcher, Nicolas Bonfante, Nicolas Brunie, Romain Bouarah, Victor Capelle, Jiajie Chen, Maxime Christ, Caroline Collange, Quentin Corradi, Orégane Desrentes, Jérémie Detrey, Antonin Dudermel, Fabrizio Ferrandi, Nicolai Fiege, Luc Forget, Martin Hardieck, Valentin Huguet, Kinga Illyes, Matei Istoan,

Mioara Joldeş, Johannes Kappauf, Cristian Klein, Marco Kleinlein, Kilian Klug, Jonas Kühle, Keanu Kullmann, Louis Ledoux, Jean Marchal, Antoine Martinet, Konrad Möller, Raul Murillo, Annika Oeste, Bogdan Pasca, Bogdan Popa, Xavier Pujol, Guillaume Sergent, Viktor Schmidt, David Thomas, Radu Tudoran, Alvaro Vasquez, Anastasia Volkova.

Finally, special thanks to all the people who have spent some of their time to review part of this book:

Hatam Abdoli, Noah Bertholon, Andreas Böttcher, Romain Bouarah, Maxime Christ, Quentin Corradi, Orégane Desrentes, Christophe de Dinechin, Silviu Filip, Luc Forget, Robin Green, Tobias Habermann, Agathe Herrou, Michael Mecik, Jean-Michel Muller, Raymond Nijssen, Pierre-Yves Piriou, and Tanguy Risset.

Contents

1	Introduction	1
1.1	Computer Arithmetic	1
1.2	Trends in Digital Technology	2
1.2.1	The Dark Silicon Era	3
1.2.2	The Dawn of the Reconfigurable Computer	4
1.3	Beyond Traditional Arithmetic Operators	7
1.4	Opportunities of Application-Specific Arithmetic	8
1.4.1	Operator Specialization	9
1.4.2	Operator Fusion	10
1.4.3	Function Approximation	12
1.4.4	Resource Sharing	14
1.4.5	Target-Specific Optimizations	15
1.5	General Design Principles for Application-Specific Arithmetic	16
1.5.1	Parameterize	16
1.5.2	Compute Just Right (Last-Bit Accuracy)	17
1.5.3	Expose the Design Space	19
1.5.4	Do Not Write Operators, Write Generators	19
1.5.5	Generate the Test Bench Along with the Operator	20
1.6	Organization of This Book	20
1.7	Support Software: The FloPoCo Project	22
1.8	Other Books on Computer Arithmetic	22
1.8.1	General Computer Arithmetic	23
1.8.2	Arithmetic for FPGAs	23
1.8.3	Other Specialized Books	24
1.8.4	Approximate Computing	25
1.9	Notations Used in This Book	26
References		29
2	Number Formats	33
2.1	Representing Integers in Binary	34
2.1.1	Binary Representation of Positive Integers	34

2.1.2	Signed Integers in Two's Complement Representation	35
2.1.3	Sign-Magnitude Representation	37
2.1.4	Conversion Between Sign-Magnitude and Two's Complement	38
2.2	Fixed-Point Binary Representations	38
2.2.1	Unsigned Fixed-Point Binary Representation	38
2.2.2	Two's Complement Fixed-Point Binary Representation	41
2.2.3	Conversion to a Wider Format: Sign Extension	42
2.2.4	Conversion to a Narrower Format: Overflow and Rounding	42
2.2.5	Alternative Notations for Fixed-Point Formats	43
2.3	High-Radix Binary Representation	44
2.4	Redundant Positional Number Systems	46
2.5	Binary Floating-Point Formats	48
2.5.1	The IEEE 754 Formats	48
2.5.2	Emerging Non-standard Formats for Machine Learning	52
2.5.3	A Simplified Floating-Point Format	54
2.5.4	Respective Motivations of IEEEfloat and Nfloat	55
2.5.5	Non-binary Floating-Point Formats	56
2.6	Logarithmic Number Systems	56
2.6.1	An Example LNS Format	57
2.6.2	LNS Operations	58
2.6.3	LNS in Arbitrary Base	59
	References	60
3	Computing Just Right: Accuracy Specification and Error Analysis	65
3.1	Accuracy of an Operator	66
3.1.1	Absolute and Relative Error	67
3.1.2	Unit in the Last Place (ulp)	67
3.1.3	Rounding to the Nearest	68
3.1.4	Truncation Versus Rounding to the Nearest	70
3.1.5	Rounding and Overflow Specification in Fixed-Point Libraries	70
3.1.6	Half-Unit Biased (HUB) Format	71
3.1.7	Rounding to the Nearest When an Approximation Is Involved	73
3.1.8	Faithful Operators and Last-Bit Accuracy	76
3.1.9	Last-Bit Accuracy Versus Rounding to the Nearest	77
3.2	Using the Format to Specify the Accuracy	77
3.3	Designing Accurate Operators	78
3.3.1	Parametric Designs	79

3.3.2	Error Analysis	80
3.3.3	Error Budget and Design-Space Exploration	83
3.4	Now Go Divide and Conquer	84
	References	85
4	Field Programmable Gate Arrays	87
4.1	Basic Logic Elements	89
4.1.1	Look-Up Tables and Flip-Flops	89
4.1.2	Basic Logic Elements of AMD FPGAs	91
4.1.3	Basic Logic Elements of Intel FPGAs	93
4.1.4	Comparison of BLEs in Commercial FPGAs	94
4.2	DSP Blocks	94
4.3	Memory Elements	95
4.3.1	Distributed Memory	96
4.3.2	Shift Register LUTs	96
4.3.3	Block Memory	98
	References	99

Part I Revisiting Classic Arithmetic

5	Fixed-Point Addition	103
5.1	Fixed-Point Considerations	104
5.1.1	Unsigned Integer Addition	104
5.1.2	Signed Integer Addition	105
5.1.3	Fixed-Point Addition	105
5.2	Addition Basics	106
5.2.1	The Full Adder and Half Adder Cells	106
5.2.2	Ripple-Carry Addition	108
5.2.3	Addition of Signed Numbers in Two's Complement	108
5.2.4	Basic Subtraction	109
5.2.5	Reconfigurable Adder/Subtractor	110
5.2.6	Carry-Save Addition	110
5.3	Fast Adders for VLSI	112
5.3.1	Switched Ripple-Carry Adder	113
5.3.2	Carry-Select Adder	114
5.3.3	Recursive Carry-Select Adder (Conditional-Sum Adder)	115
5.3.4	Carry-Lookahead Adder	116
5.3.5	Recursive Carry-Lookahead Adder	117
5.3.6	Parallel Prefix Adders	118
5.3.7	Compound Fast Adder	123
5.3.8	Fast Absolute Difference Operator	124
5.3.9	Fast Compound Triple Sum	124
5.4	Adders on FPGAs	125

5.4.1	Addition Support on AMD FPGAs	126
5.4.2	Addition Support on Intel FPGAs	128
5.4.3	Merging Additional Functionality in an Adder	128
5.4.4	Ternary Adders on FPGAs	129
5.4.5	Reconfigurable Adder/Subtractor	133
5.5	Fast Adders on FPGAs	134
5.5.1	Pipelined Adders	135
5.5.2	Fast Combinatorial Adders	137
	References	140
6	Fixed-Point Comparison	145
6.1	Introduction	145
6.1.1	Fixed-Point Considerations	146
6.1.2	Area and Delay Considerations	146
6.2	Basic Binary Tree Comparison	147
6.3	FPGA-Specific Implementations	148
6.3.1	Exploiting Fast Carry Logic	148
6.3.2	Two-Level Tree of Fast Carry Logic	148
6.3.3	Comparing to a Constant	149
	References	150
7	Sums of Weighted Bits	151
7.1	Definitions and Motivation	152
7.1.1	A Multiplier Using a Compressor Tree	153
7.1.2	From Bit Arrays to Bit Heaps	157
7.1.3	Bit Heaps for Portable Application-Specific Arithmetic	160
7.2	Expressing a Computation as a Bit Heap	161
7.2.1	Managing Constant Bits	161
7.2.2	Managing Signed Numbers	162
7.2.3	Algebraic Optimizations	163
7.2.4	Truncating a Bit Heap for Last-Bit Accuracy	167
7.3	Compressor Tree Synthesis	175
7.3.1	Basic Compression Algorithms	176
7.3.2	A Bestiary of Compressors	180
7.3.3	Compressor Tree Synthesis Using Arbitrary Compressors	186
7.3.4	Some Remarks on the Final Adder	193
7.4	Experimentations	194
	References	198
8	Fixed-Point Multiplication	203
8.1	A Functional Point of View	204
8.1.1	Integer Multipliers	204
8.1.2	Fixed-Point Numbers	206

8.1.3	Error Analysis Involving Multipliers	209
8.1.4	Summary: Multipliers for Computing Just Right	211
8.2	Overview of Multiplier Construction	213
8.3	Partial Product Generation and Sign Management	214
8.3.1	Signed Multiplication in Radix-2	215
8.3.2	Radix-4 Multiplication Using Booth Encoding	215
8.3.3	Higher Radix Multiplication	221
8.4	Multiplier Construction for FPGAs	222
8.4.1	Using a Single DSP for Multiple Multiplications	223
8.4.2	Multiplier Construction as a Tiling Problem	226
8.4.3	Solving the Tiling Problem	233
8.5	Truncated Multipliers	237
8.5.1	Plain or Booth Truncated Multipliers	237
8.5.2	Tiling for Optimal Truncated Multipliers on FPGAs	239
8.6	The Karatsuba Method	244
8.6.1	Two-Part Splitting	244
8.6.2	Subtractive Karatsuba Formula	245
8.6.3	Tile Representation	246
8.6.4	Square K-Part Splitting	246
8.6.5	Recursive Karatsuba	247
8.6.6	Generalized Karatsuba Formula	249
8.6.7	Karatsuba with Rectangular Sub-multipliers	249
	References	255
9	Fixed-Point Division	259
9.1	Fixed-Point Division: Problem Formulation	260
9.1.1	Format Considerations for Unsigned Integer Division	262
9.1.2	Format Considerations for Signed Integer Division	262
9.1.3	Format Considerations for Fixed-Point Division	262
9.1.4	Division of Normalized Floating-Point Significands	263
9.2	Digit-Recurrence Algorithms	264
9.2.1	Schoolbook Integer Division in Binary (Restoring Division)	266
9.2.2	General Radix- β Formalization of Integer Division	268
9.2.3	General Radix- β Formalization for the Division of Normalized Significands	271
9.2.4	Using Redundant Digit Sets	272
9.2.5	Initialization and Termination with Redundant Digits	279

9.2.6	Conversion of the Quotient from Redundant Form to Binary	280
9.2.7	High-Radix Integer Division	282
9.2.8	Prescaling	283
9.2.9	Speeding Up the Partial Remainder Computation ..	286
9.2.10	A Case Study for Low-Latency Double Precision Division	288
9.3	Division Using Multiplication by the Reciprocal	288
9.3.1	Error Analysis for a Faithfully Rounded Quotient out of a Reciprocal Approximation	289
9.3.2	Reciprocal Using a Generic Approximation Method	291
9.3.3	Reciprocal Using Newton-Raphson Iteration	291
9.4	Division Using Quadratic Series Expansion	294
9.5	Other Equivalent Multiplicative Methods	299
9.5.1	Multiplicative Normalization (Goldschmidt's Algorithm)	300
9.5.2	Higher-Order Householder Methods	301
9.5.3	Ad hoc Evaluation of Series Expansion	301
9.6	Square Root, the Little Sister of Division	302
	References	303
10	Shifters and Leading Bit Counters	307
10.1	Shifters	307
10.1.1	Avoiding Bidirectional Shifters	308
10.1.2	Parameters of a Generic Shifter Generator	309
10.1.3	Architecture of an Exact Full Shifter	312
10.1.4	Barrel Shifter	314
10.1.5	Barrel Shifter with Early Sticky Bit Computation ..	314
10.2	Variations on Leading Zero Counters	316
10.2.1	Naive LZC Architecture	318
10.2.2	Logarithmic-Time LZC Architectures	320
10.3	Normalizer (Combined LZC + Shifter)	323
10.4	To Read Further	325
	References	326
11	Basic Floating-Point Operators	329
11.1	Floating-Point Addition and Subtraction	331
11.1.1	General Considerations and Terminology	331
11.1.2	Baseline Addition Architecture	336
11.1.3	From Baseline Adder to IEEE 754 Compliance	339
11.1.4	Dual-Path Architectures and Other Speculative Techniques	340

11.2	Floating-Point Multiplication	343
11.2.1	Baseline Multiplication Architecture	343
11.2.2	From Baseline Multiplier to IEEE 754 Compliance	345
11.2.3	Faithful Floating-Point Multiplier	346
11.2.4	Injection Rounding to Speed Up a Floating-Point Multiplier	346
11.3	Floating-Point Division	347
11.3.1	Baseline Division Architecture	347
11.3.2	From Baseline Divider to IEEE 754 Compliance	350
11.3.3	Faithful Floating-Point Division	351
11.3.4	Correct Rounding Out of a Faithfully Rounded Quotient	351
11.4	Floating-Point Square Root	353
11.4.1	Baseline Square Root Architecture	353
11.4.2	Faithful Floating-Point Square Root	355
11.4.3	From Baseline Square Root to IEEE 754 Compliance	356
11.4.4	Correct Rounding Out of Faithful Rounding for Square Root	356
11.5	Floating-Point Comparison	357
11.5.1	Specification of Floating-Point Comparison	357
11.5.2	Implementation of a Floating-Point Comparator	359
11.5.3	Specializations of a Floating-Point Comparator	360
	References	360

Part II Operator Specialization

12	Multiplication by Constants	365
12.1	Shift-and-Add Integer Constant Multiplication	366
12.1.1	Signed Digit Representation	368
12.1.2	Formalization of the Shift-and-Add SCM Problem	369
12.1.3	Minimal-Adder Constant Multiplication by Graph Enumeration	370
12.1.4	Minimal-Adder Constant Multiplication by Using ILP	373
12.1.5	Minimal-Adder Constant Multiplication Using Ternary Adders	378
12.1.6	Minimizing Logic Resources Instead of Number of Adders	380
12.2	Integer Constant Multiplication Using Tables	383
12.2.1	Tabulated Constant Multipliers	383
12.2.2	The Table-and-Addition KCM Algorithm	384
12.3	Multiplication of a Fixed-Point Number by a Real Constant	385
12.3.1	Tabulated Perfectly Rounded Constant Multipliers	386

12.3.2	Faithful Fix-by-Real Using Shift-and-Add	387
12.3.3	Faithful Fix-by-Real KCM Algorithm	390
12.4	Multiplication by a Rational Constant	395
12.4.1	On the Periodicity of the Binary Representation of Rational Numbers	396
12.4.2	Periodical Shift-and-Add Graphs	397
12.4.3	Table-Based Multiplication by Rational Constants	401
12.5	Multiplication by Multiple Constants	401
12.5.1	Multiple Constant Multiplication Using Shift-and-Add	402
12.5.2	Table-Based Multiple Constant Multiplication	407
12.5.3	Sum of Constant Products, Integer Case	407
12.5.4	Constant Matrix-Vector Multiplication	410
12.5.5	Table-Based Sum of Products of Fixed-Point Inputs by Real Constants	412
12.6	Other FPGA-Specific Techniques	414
12.6.1	Constant Multiplication Using DSP Blocks on FPGAs	414
12.6.2	Reconfigurable Constant Multiplication	416
12.7	Conclusion: Choosing the Best Constant Multiplication Technique in a Given Context	418
	References	418
13	Division by Constants	427
13.1	Multiplying by the Reciprocal	428
13.2	Linear Table-Based Architecture	429
13.2.1	Radix- 2^k Representation	430
13.2.2	Algorithm	430
13.2.3	Iterative or Unrolled Implementation of the Basic Recurrence	432
13.2.4	Cost Evaluation of LinArch	432
13.2.5	FPGA-Specific Remarks	434
13.3	Parallel Division	434
13.4	Remainder-Only or Quotient-Only	435
13.4.1	Reciprocal Method Outputting Only the Quotient	435
13.4.2	Remainder-Only Variant of LinArch and BTCD	436
13.5	Composite Division	437
13.5.1	Algorithm	437
13.5.2	Architecture	438
13.5.3	Results and Discussions	438
	References	440
14	Fixed-Point Squares, Cubes, and Other Integer Powers	443
14.1	Generalities	443
14.2	Squarers	444
14.2.1	Basics	444

14.2.2	High-Radix Square	445
14.2.3	Booth Recoding	446
14.2.4	Truncated Squarers	446
14.2.5	Squareer-Specific Tiling for FPGAs	447
14.3	Cubes and Higher Powers	449
14.3.1	Direct Algebraic Expression	449
14.3.2	Computing Powers by Squaring and Multiplying	450
14.3.3	Approximate Fixed-Point Powers	451
	References	451
15	Specialization and Fusion of Floating-Point Operators	453
15.1	Floating-Point Constant Multiplication	453
15.1.1	Floating-Point Multiplication by a Power of Two	454
15.1.2	Baseline Faithful Floating-Point Multiplier by a Constant	455
15.1.3	Correctly Rounded Multiplication by a Floating-Point Constant	457
15.1.4	Correctly Rounded Multiplication by a Real Constant	459
15.1.5	Correct Rounding of the Floating-Point Multiplication by a Rational Constant	461
15.1.6	Subnormal Handling	462
15.2	Floating-Point Division by a Small Integer Constant	464
15.2.1	Baseline Operator for Normalized Inputs	464
15.2.2	Subnormal Handling	467
15.3	Floating-Point Squares, Cubes, and X^p	467
15.4	Floating-Point Addition of Positive Terms	468
15.5	Combined Floating-Point Sum and Difference	469
15.6	Fused Multiply and Add, Other Sums of Products	469
15.7	Floating-Point Optimizations in an HLS Context	470
	References	472

Part III Generic Methods for Fixed-Point Function Approximation

16	Generalities on Fixed-Point Function Approximation	477
16.1	Defining Domain and Range	479
16.2	Discretization Issues	480
16.2.1	Range Issues and Their Solutions	481
16.2.2	Monotonicity Issue	485
16.3	Some Classes of Numerical Functions	486
16.3.1	Algebraic Functions	486
16.3.2	Elementary and Special Functions	487
16.4	A First Generic Approximator: Simple Tabulation	488
16.4.1	Tabulating Precomputed Function Values in a ROM	489
16.4.2	Actual Cost of a ROM	491
	References	492

17	Function Evaluation Using Tables and Additions	497
17.1	Lossless Differential Table Compression	497
17.1.1	Applicability	498
17.1.2	The Parameter Space of LDTC	499
17.1.3	The LDTC Optimization Algorithm	500
17.1.4	Cost Function	501
17.1.5	Evaluation and Observations	502
17.2	Multipartite Methods	503
17.2.1	Applicability	503
17.2.2	The Basic Bipartite Method	504
17.2.3	Exploiting Symmetry	507
17.2.4	From Bipartite to Multipartite: Splitting the TO	508
17.2.5	Error Analysis	511
17.2.6	Computing the Sizes of the TO_i	516
17.2.7	Filling the Tables Using HUB Format	517
17.2.8	Errorless Compression of the TIV	518
17.2.9	Putting It All Together: The Multipartite Optimization Algorithm	519
17.2.10	The Issue of Non-Monotonicities	521
17.2.11	Toward ILP-Optimized Multipartite Architectures	522
17.2.12	Conclusion: Multipartite Architectures Are Close to Optimal Among Order-One Methods	523
17.3	Other Table-and-Addition Methods	523
17.3.1	Addition-Table-Addition Methods	523
17.3.2	Partial Product Arrays	526
	References	527
18	Polynomial-Based Architectures for Function Evaluation	529
18.1	A Primer on Polynomial Approximation	530
18.1.1	Taylor Approximation	531
18.1.2	Remez Minimax Approximation	533
18.1.3	Relationship Between Degree and Accuracy	535
18.1.4	Relationship Between Interval Size and Accuracy	535
18.1.5	Machine-Efficient Polynomial Approximation	536
18.1.6	Summing Up	538
18.1.7	Polynomials Are a Special Case of Rational Fractions	539
18.2	Generic Range Reduction Techniques	541
18.2.1	Range Reduction by Uniform Piecewise Splitting	541
18.2.2	Interpolation	547
18.2.3	Logarithmic Piecewise Splitting	548
18.2.4	Hierarchical Segmentation	551
18.2.5	Arbitrary Dichotomy-Based Segmentation	552
18.2.6	Discussion	553

18.3	Hardware Polynomial Evaluation	554
18.3.1	Horner Evaluation	554
18.3.2	Parallel Evaluation	558
18.3.3	Other Polynomial Evaluation Techniques	561
18.4	Putting It All Together: Generation of Polynomial Approximators	562
18.4.1	Overall Error Analysis and Error Budget	563
18.4.2	A Basic Polynomial Approximator Without Range Reductions	564
18.4.3	A Polynomial Approximator with Uniform Piecewise Range Reduction	565
18.4.4	Combined Approximation and Rounding Optimization Using integer linear programming (ILP)	566
18.5	Floating-Point Architectures Based on Fixed-Point Polynomials	568
	References	569
19	Digit Recurrence for Algebraic Functions	573
19.1	Digit-Recurrence Square Root	574
19.1.1	Range Reduction for Floating-Point and Fixed-Point Binary	574
19.1.2	Generic Radix- β Formulation	575
19.1.3	Simple Binary Restoring Square Root	576
19.1.4	Binary Non-restoring Square Root	578
19.1.5	Exploiting Redundant Number Systems in the Square Root	580
19.2	Cube Root	588
19.3	2D and 3D Euclidean Norms	589
19.4	The E-Method for Evaluating Rational Functions	591
19.4.1	Digit Recurrence	591
19.4.2	Rational Approximation Compatible with the E-Method	593
	References	594
Part IV Example Composite Operators		
20	Fixed-Point Sine and Cosine	599
20.1	Mathematical Background	599
20.2	Fixed-Point Specification	602
20.2.1	Binary Angles	602
20.2.2	Scaled Output	603
20.2.3	Quick Overview of the Algorithms Compared in This Chapter	604
20.3	Argument Reduction	605
20.4	CORDIC Computing Just Right	605
20.4.1	Description of the Algorithm	605

20.4.2	Overall Error Budget	609
20.4.3	Approximation Error and Number of Iterations	610
20.4.4	Rounding Errors	610
20.4.5	Reduced Z_i Datapath	613
20.4.6	Replacing Half of the CORDIC Iterations with a Small Multiplier	613
20.5	An Architecture Based on Tables and Multipliers	614
20.5.1	Algorithm	615
20.5.2	Rounding Error Analysis and Implementation Details	616
20.5.3	Architectural Details	618
20.5.4	Computing the Number of Guard Bits	619
20.5.5	Special Cases for Small Sizes	620
20.6	Architecture Using a Generic Polynomial Evaluator	620
20.7	Comparison and Discussion	620
	References	621
21	Floating-Point Accumulation and Sum of Products	623
21.1	Motivation and Parametrization	625
21.1.1	Exact Floating-Point Sum	625
21.1.2	Exact Floating-Point Sum of Products	626
21.1.3	Kulisch's Universal Exact Accumulator	627
21.1.4	Application-Specific Parameterization	628
21.2	Accumulator Architectures	630
21.2.1	Exact Floating-Point Multiplier	630
21.2.2	Simple Accumulator Architectures for Small Precisions	632
21.2.3	High-Radix Carry-Save Architecture	635
21.2.4	Conversion of the Accumulator Back to Floating Point	636
21.3	Cost, Speed, and Accuracy	638
21.4	To Probe Further	639
	References	639
22	Floating-Point Exponential	641
22.1	Mathematical Background	641
22.2	Number Formats and the Exponential Function	642
22.2.1	Fixed Point Versus Floating Point	642
22.2.2	Fixed-Point-In, Floating-Point-Out Version	643
22.2.3	Input Filtering	643
22.2.4	A First Architecture: Direct Tabulation	645
22.3	Architecture for Floating-Point Range Reduction	646
22.3.1	Overview	646
22.3.2	Accuracy Considerations	647

22.3.3	Fixed-Point Conversion	648
22.3.4	Computation of the Tentative Exponent	649
22.3.5	Computation of Y	650
22.3.6	Computation of e^Y	651
22.3.7	Normalization and Rounding	651
22.3.8	Overall Error Analysis	651
22.4	Fixed-Point-In, Floating-Point Out Exponential	654
22.5	Fixed-Point Computation of e^Y	654
22.5.1	Using Large Tables and Square Multipliers	655
22.5.2	First-Order Architecture	659
22.5.3	Polynomial Approximation for Large Precisions	660
22.5.4	FPGA-Specific Remarks	661
22.5.5	Iterative Computation of e^Y Using Small Tables and Rectangular Multipliers	662
22.5.6	To Read Further	663
	References	664

Part V Application Domains

23	Arithmetic in the Design of Linear Time-Invariant Filters	669
23.1	An Introduction to Discrete-Time Signals and Filters	670
23.1.1	Discrete-Time Signals	670
23.1.2	Elementary Operations on Discrete-Time Signals	670
23.1.3	Discrete-Time Systems or Filters	671
23.1.4	Linear Time-Invariant Filters	672
23.1.5	Specifying LTI Filters by Constant-Coefficient Equations	676
23.1.6	Abstract Filter Structures Versus Finite-Precision Circuits	681
23.1.7	Filters Directly Defined by Their Coefficients	681
23.1.8	Filters Defined in the Frequency Domain	682
23.1.9	Filter Design: From a Frequency Specification to a Time-Domain Architecture	685
23.2	An Arithmetic Approach to the Implementation of LTI Filters	687
23.3	Hardware Digital Filter Faithful to an LTI Filter	688
23.3.1	Worst-Case Peak Gain of an LTI Filter	689
23.3.2	Interface Considerations	690
23.3.3	Overall Error Analysis of the Implementation of a Direct-Form LTI Filter	691
23.3.4	Error Amplification in the Feedback Loop	692
23.3.5	Putting All Together	694
23.4	Hardware Digital Filter Faithful to a Frequency Specification	695
23.4.1	Formal Definition	695
23.4.2	Hardware Filter Design as an Optimization Problem .	695

23.4.3	State of the Art in the Design of Filters Faithful to a Frequency Specification	697
23.4.4	Frequency Constraints for Linear-Phase FIR Filters in ILP	698
References	702
24	Arithmetic for Deep Learning	707
24.1	Neural Network Overview	708
24.1.1	Basic Neural Network Structure	708
24.1.2	Activation Functions	712
24.1.3	Topology and Hyperparameters	713
24.1.4	Training and Using a Neural Network	713
24.1.5	Training Metrics	715
24.2	Convolutional Neural Networks	715
24.2.1	Convolutional Layers	716
24.2.2	Pooling Layers	719
24.2.3	Batch Normalization	720
24.2.4	Passthrough Layer	721
24.2.5	Softmax Layer	721
24.2.6	An Example CNN	722
24.3	Number Formats	722
24.3.1	Number Formats for Inference	724
24.3.2	Number Formats for Training	729
24.4	Training	730
24.4.1	The Training Data	730
24.4.2	Training by Backpropagation	731
24.4.3	Training and Activation Functions	733
24.4.4	Quantization-aware training (QAT)	733
24.5	Implementation Aspects	734
24.5.1	Architectures for Inference	734
24.5.2	Fast Convolution Algorithms	737
24.6	Case Studies	741
24.6.1	Google's TPU Family	741
24.6.2	Binary CNN Architecture	743
24.6.3	AddNet: FPGA-Specific Optimization of the Multipliers	744
24.6.4	Unrolling a Ternary CNN	746
24.6.5	LogicNets	750
References	751
A	Custom Arithmetic Datapath Design with FloPoCo	761
A.1	History of the FloPoCo Project	761
A.2	FloPoCo From a User Point of View	762
A.2.1	Overview	762
A.2.2	More on Parameters	763

A.2.3	Do Not Trust FloPoCo, the Test Bench Is Included	764
A.2.4	Obscure Branches and Code Attics	766
A.2.5	Automatic Pipelining, the User Point of View	766
A.3	FloPoCo for Arithmetic Designers	769
A.3.1	Operators and Instances	770
A.3.2	The Target Class Hierarchy	771
A.3.3	Automatic Pipelining	772
A.3.4	The BitHeap Framework	775
References	777
B	Optimization Using Integer Linear Programming	781
B.1	Linear Programming Problems	782
B.1.1	A First Example Problem	783
B.1.2	Integer, Binary, and Mixed-Integer Linear Programming	783
B.2	A Tutorial on Using ILP Solvers	784
B.2.1	Using Stand-Alone Tools	785
B.2.2	ILP Solvers Embedded Within Other Scripting Language	787
B.2.3	ScaLP, a Common C++ Interface to ILP Solvers	787
B.3	Practical Problem Modeling with ILP	787
B.3.1	Using Boolean Variables to Model Decision Problems	788
B.3.2	Translating Boolean Relations into Constraints	788
B.3.3	Indicator Constraints	793
B.3.4	Splitting Integers into Their Binary Representation	793
B.3.5	Counting Leading and Trailing Zeros	794
B.4	Addressing Limitations of ILP Solvers	795
B.4.1	Run-Time	795
B.4.2	Numerical Instabilities	796
References	796
Index	799



1

CHAPTER 1

Introduction

*Mathematics is the queen of the sciences,
and arithmetic is the queen of mathematics.*

Carl Friedrich Gauß

*That arithmetic is a menial task is proved
by the fact that it is the only one that can be performed by a machine.*

Arthur Schopenhauer

1.1 Computer Arithmetic

Computer arithmetic is the art of representing and processing numbers in a machine. It has its roots in the abacus of ancient times and in the mechanical calculator built by Schickhard, Pascal, Leibniz, and others during the Renaissance. It has also been at the core of electronic computers since the dawn of this technology in the 1940s.

Computer arithmetic is deeply rooted in technology. For instance, when technology was based on humans' head and fingers, numbers were represented in radix 10 or 60. Western abacuses, the Pascaline, the Arithmometer, and the Eniac were decimal calculators. In the mid-1930s, however, Konrad Zuse understood that machines would be more efficient when computing in binary [Zus84; Cer90; Roj97]. The key idea was that it was much easier to work with two states, on and off. After two years of attempts to process binary numbers mechanically, Zuse turned to relay-based switching, and

later to vacuum tubes, and the idea remained relevant with transistor-based switching, the core technology behind current computers.

During the electronic computer era, technological progress has often driven evolutions in the computer arithmetic domain. One illustrative example is the use of memory when evaluating elementary functions (exponential and logarithm, trigonometric, etc.). In the late 1980s, as memory sizes doubled every other year following Moore's law, it became relevant to design algorithms that could replace long computations with large tables of precalculated values [Tan89; Tan90; GB91; Tan92]. Such algorithms dominated the state of the art for three decades. But what technology gives, it can also take back. Even though memory size still increases, memory access time has become the main performance bottleneck in current multicore computers. For this reason, table-free algorithms are now considered more efficient in many cases [CHT02]: it is often faster and more energy-efficient to compute tens of additions and multiplications than to perform one memory lookup.

Digital technology is still evolving. How should computer arithmetic adapt to current and upcoming changes?

One answer proposed in this book is look beyond traditional computer arithmetic, long focused on the four basic operations and their monolithic, one-size-fits-all implementations in the arithmetic units at the core of microprocessors. The object of this book is the systematic study of non-standard arithmetic units, built and optimized for specific applications: what we call application-specific arithmetic.

The remainder of this chapter first justifies this claim and then sketches the scope and potential of application-specific arithmetic.

1.2 Trends in Digital Technology

For five decades, the evolution of technology has been well summarized by Moore's law, which states that the number of transistors that can be integrated on an economically viable chip doubles every other year. This law has remained surprisingly accurate over this long period of time, partly because it has become a self-fulfilling prophecy on which the economics of a major part of the digital sector relies. The International Technology Roadmap for Semiconductors (ITRS) has made sure that we get smaller transistors (or equivalently more transistors per chip) as long as fundamental physical limits are not reached.

However, the circuit-level performance brought by smaller transistors is no longer what it used to be. Until the end of the twentieth century, each technology generation brought smaller transistors that would switch faster and consume less power (this is referred to as Dennard scaling). Thus, processors were more and more powerful and clocked higher and higher. How-

ever, there is a limit on the amount of heat per square centimeter that can be dissipated off a chip, and this limit (the “power wall”) was reached around 2004. From there on, transistors could still be made smaller with each generation, but their practical operational frequency reached a plateau. This prompted the switch to multicore processors as the most efficient means to harness increased transistor densities.

Unfortunately, in the upcoming technology generations, the increase of transistor density will entail a further increase in power density. Roughly speaking, every two years, transistor density is multiplied by two, but the power dissipation of each transistor is only reduced by a factor 1.4 [Tay12]. The net effect is that we can no longer afford to operate 100% of a chip all the time. A certain percentage must be kept switched off (or dark, hence the term *dark silicon* [Mer09]). Even worse, this percentage is now increasing exponentially with each technology generation.

1.2.1 The Dark Silicon Era

Taylor [Tay12] reviews architectural solutions to the dark silicon problem, some of which are relevant to computer arithmetic. The main one is to complement the usual, universal processor cores with specialized coprocessors that offer only one functionality, but with much higher efficiency. Such specialized coprocessors may be arithmetic components or may be dedicated to higher-level functionality such as cryptography, signal processing for software-defined radio, face detection, fingerprint recognition, etc. Each of these high-level coprocessors will itself require application-specific arithmetic, for instance, a multiplier by $\log(2)$ in a coprocessor computing a floating-point exponential or a low-precision $\text{atan}(x/y)$ in a signal-processing pipeline.

Let us open a parenthesis about the recent history of the division operation in microprocessors, to illustrate that this solution is already at work. The design of hardware dividers has been the subject of much research and is very well covered by other books [EL94; EL04]. In 1997, a paper by Oberman and Flynn [OF97a] investigated in detail the relationship between divider latency and system performance. They observed that only 3% of floating-point operations are divisions in the SPECFP92 benchmark suite, but that these few divisions could account for up to 40% of the latency if a low-area, long-latency divider is used. The conclusion was that low latency hardware dividers were needed in order to achieve the best performance. This was in line with the prevalent IA32 instruction set, which has included division instructions and division hardware since the original 8086 processor (integer division) and 8087 coprocessor (floating-point division). Despite this, a few years later, at the turn of the century, the IA64 instruction set was designed by HP and Intel without division hardware and even without a division

instruction. The argument was the following: it was possible to build very efficient division in software thanks to a new instruction, the Fused Multiply-Add (FMA) [Mar00]. Therefore, the silicon area occupied by a hardware divider could be much better spent, for instance, on a second FMA, which is a more generally useful operator than a divider.

However, we are now seeing a comeback of hardware dividers in processors [Bru18; Bru22]. The reason for that is, again, power consumption (and not performance as in 1997 [OF97a]). A nicely pipelined hardware divider will be much more power-efficient at computing a division than software: the latter needs to read and decode instructions, move data from and to the register file, etc. Even if a hardware divider would not be faster than the equivalent software (and it is), it would increase the overall performance by freeing some power budget to be used by other computations. Therefore, it seems a worthwhile use of dark silicon.

What function is next? Square root is very comparable to division in terms of algorithms and complexity. However, still according to [OF97a], it accounts for less than 0.33% of floating-point instructions. Then, we may see a regain of interest in implementations of floating-point elementary functions such as exponential, logarithm, and trigonometrics. Some GPUs (graphics processing units) already provide hardware acceleration for the most common of these functions [OS05].

Another way to put dark silicon to use, according to [Tay12], is to use more area to buy energy. For instance, since power dissipation is proportional to the square of the frequency, two copies of a functional block, each clocked to half the frequency, will actually consume half the power. Here, what is needed is not a new operator, but a performance variation on an existing one. This book will also attempt to cover this performance space.

We will conclude this section about dark silicon with three quotes. The first two come from Don Burger's keynote talk at the HIPEAC 2013 conference:

“The end of Moore’s Law doesn’t mean the end of progress”, and “nothing in our careers has been as fundamental as this transition.”

The third one is from the conclusion of [Tay12]: “Although silicon is getting darker, for researchers the future is bright and exciting.”

The intent of this book is to bring some of this excitement in the computer arithmetic domain.

1.2.2 The Dawn of the Reconfigurable Computer

A field programmable gate array (FPGA) is an integrated circuit designed to emulate arbitrary digital circuits. Technically, its structure is that of a grid of configurable logic functions, embedded in a fabric of configurable routing channels (see Fig. 1.1). The word *configuration* describes the process of

assigning a function to the field programmable gate array (FPGA). After configuration, the FPGA behaves as the circuit it has been configured for. The configuration of an FPGA is held in static registers and can be changed arbitrarily. More details will be given in Chap. 4.

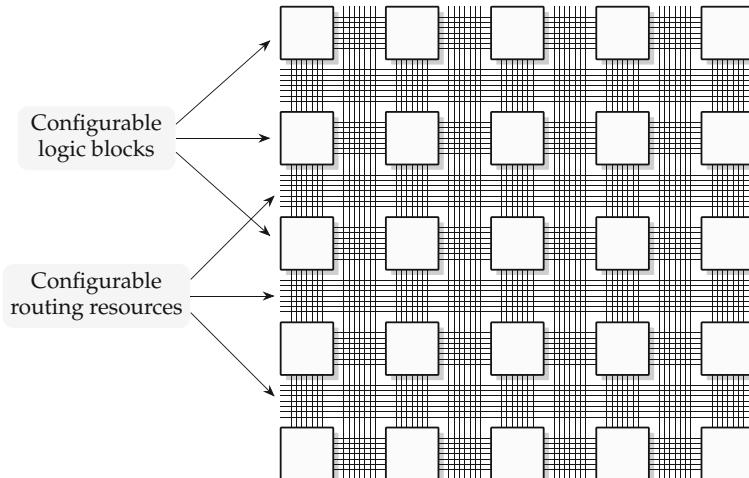


Fig. 1.1 Simplified overview of FPGA structure.

This configurability has a cost: A circuit emulated in an FPGA typically performs one order of magnitude slower than the same circuit cast directly to silicon [KR07]. This FPGA *performance gap* is mostly due to the reconfigurable routing; wires are longer, and information has to pass through the switches that ensure the reconfigurability. On a positive note, FPGAs have not yet hit their power wall. This is due to their lower operating frequency, combined with a vast area for the reconfigurable routing that dissipates more heat than it generates.

FPGAs were initially designed for rapid prototyping (emulating a logic design before committing it to silicon) and as a flexible replacement to programmable logic arrays. However, they evolved to universal computing machines. For instance, they can emulate a conventional computer, but also other programming models, such as von Neumann's cellular automata or data-flow machines. They were therefore soon used as "reconfigurable computers." At the time of writing this book, several companies successfully sell FPGA-based computing, and FPGA-accelerated cloud services are commercially available.

How can an FPGA compete with a processor considering its performance gap? The first answer is that a processor is also inherently quite inefficient

due to its sequential programming model. A typical processor cycle consists of fetching an instruction, decoding it, reading the operands from registers, executing the instruction, and finally storing the result in registers. Among all these steps, only one step (execution) actually performs the computation. All the other steps are there for program and data management. Hence, an FPGA architecture that only implements the execution steps will typically be more efficient. Figure 1.2 shows an example of such an architecture for a simple computation. There is no program management necessary: the various operators are laid out on the FPGA fabric. There are registers (typically between each operator, or even inside operators in a pipelined design), but these registers are statically assigned and therefore without the address decoding logic found in a processor's register file.

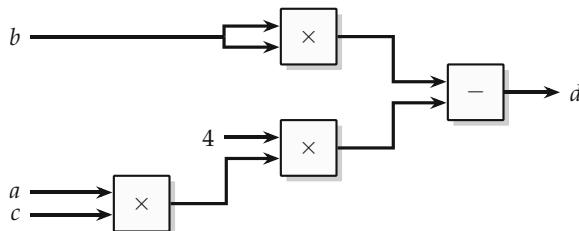


Fig. 1.2 Dataflow computation of $d = b^2 - 4ac$.

A second way an FPGA can outperform a processor despite its performance gap is a better use of parallelism. If a stream of data is available at the input of a dataflow architecture such as that of Fig. 1.2, all its operators may function in parallel, in a pipeline fashion. Besides, the operators implemented in this architecture exactly match the requirements of the application. There is also a lot of parallelism in a processor: single instruction, multiple data (SIMD) parallelism (as in the MMX/SSE/AVX instructions of Pentium-like processors), superscalar parallelism, multicore parallelism, and Symmetric Multi-Threading (SMT). However, the amount of operators is fixed. It has been carefully engineered to best match the average load of a processor, but will almost always be suboptimal for a given application. Consider, for instance, the computation of $d = b^2 - 4ac$ in floating point. Here, we have more multipliers than additions. In a typical processor, the ratio of floating-point multipliers to floating-point adders is 1:1, so this computation under-uses the addition hardware. Besides, many other hardware operators are unused: the integer ones, the divider, etc.

A third, and often overlooked, way to bridge the FPGA performance gap is to tailor, as tightly as possible, the arithmetic to the application. A maximally efficient implementation would, for each of its operations, toggle and transmit just the number of bits required by the application at this point. Conventional microprocessors have a fixed, limited choice of data widths

(e.g., 32 and 64 bits). Most of the time, this will be too much – or not enough – but then the program will fail.

FPGAs, with their bit-level granularity, have the potential to get much closer to this ideal of what we call *computing just right*. Therefore, reconfigurable computing should systematically investigate, in an application-specific way, non-standard precisions, but also non-standard number systems and non-standard arithmetic operations.

The purpose of this book is to review these opportunities. Before surveying them in Sect. 1.4, we have to define precisely what we call in this book an “operator.”

1.3 Beyond Traditional Arithmetic Operators

Let us begin with an informal definition.

Definition 1.1 An arithmetic operator is the well-specified machine implementation of a mathematical function.

Here, mathematical functions include the basic arithmetic operations ($+$, $-$, \times , $/$), elementary and special functions (exponential, logarithm, trigonometric, etc.), but also much coarser operations such as matrix or tensor product, signal processing kernels such as the discrete Fourier transform (DFT), etc. Remark that a matrix product or a DFT can be defined by composition of simpler functions (addition and multiplication), just like the example used earlier $d = b^2 - 4ac$, but other functions may be defined differently, for instance, as the solution of a differential equation. We deliberately do not attempt to restrict the set of mathematical functions covered in this definition: new applications will bring new operators.

As machines must operate with finite precision, the term “well-specified” in the definition above must take into account this aspect. This important question is the subject of most of Chaps. 2 and 3, and we just illustrate it with a well-known example here. The IEEE 754 standard for floating-point arithmetic [754-19] defines number formats, for instance, binary32 (also known as single precision). For each format, it also defines *rounding functions* that transform a real number into a value of this format. The most common is *rounding to nearest with ties to even*, denoted as $\text{RN}(x)$. Then, in this standard, a floating-point addition operator $\tilde{+}$ is “well specified” as the composition (in the usual mathematical sense) of the mathematical operation $+$ with the rounding function: $x \tilde{+} y = \text{RN}(x + y)$. This is illustrated by Fig. 1.3. Of course in practice the inputs x and y to $\tilde{+}$ will be floating-point numbers themselves, but this needs not appear in the definition, as a floating-point number is also a real number.

In practice, it is also often convenient to add to the set of real numbers a handful of *special values* that capture exceptional situations. For instance, the

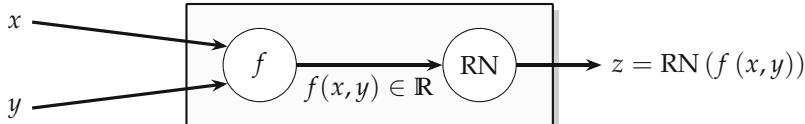


Fig. 1.3 An operator specified as the composition of a mathematical function and a rounding function.

reader may have encountered the signed infinities or the *Not a Number* values of the IEEE 754 standard. These are non-numerical special values, but the standard also defines two signed zeroes (and their relationship to the zero of real numbers). The behavior of an operator must be defined specifically for these values. It may also produce them, e.g., in case of underflow, overflow, or undefined situations.

This approach will be generalized in this book to many mathematical functions, but also many number formats, with or without special values. Besides, it will be useful to relax the notion of rounding from a function to a binary relation. For instance, for a machine number format \mathbb{F} and a positive error bound $\bar{\delta}$, we could say that $X \in \mathbb{F}$ is an acceptable rounding of $x \in \mathbb{R}$ if and only if $|X - x| < \bar{\delta}$. This idea will be refined in Chap. 3.

With such a well-specified rounding relation, the following definition is a formal refinement of Definition 1.1:

Definition 1.2 Let \mathbb{S} be a finite (possibly empty) set of special values. Let $f : (\mathbb{R} \cup \mathbb{S})^n \rightarrow \mathbb{R} \cup \mathbb{S}$ be a mathematical function extended to special values.

Let $\mathbb{F} \subset (\mathbb{R} \cup \mathbb{S})$ be the finite set of values available in a given machine format.

Let \approx be a well-defined rounding relation on $\mathbb{F} \times (\mathbb{R} \cup \mathbb{S})$.

An operator Op implements f for the format \mathbb{F} with rounding $\overset{r}{\approx}$ if and only if

$$\forall (X_1, \dots, X_n) \in \mathbb{F}^n \quad \text{Op}(X_1, \dots, X_n) \overset{r}{\approx} f(X_1, \dots, X_n) \quad .$$

We will no longer burden our reader with this overly formal definition, but each and every operator discussed in this book is designed to a specification which is an instance of Definition 1.2. In other words, it is well specified. This helps ensure correctness, of course, but it also helps optimizing for performance by providing a clear rule of the game against which algorithms and implementations may compete.

1.4 Opportunities of Application-Specific Arithmetic

There are five main techniques by which computer arithmetic can be tailored to the needs of a specific application: *operator specialization*, *operator fusion*, *function approximation*, *resource sharing*, and *target-specific optimizations*.

1.4.1 Operator Specialization

Operator specialization consists in optimizing the structure of an operator thanks to some useful property on its inputs.

First, an operator with a constant operand can often be optimized somehow:

- For application-specific integrated circuits (ASICs) or FPGAs, multiplication by a constant has been extensively studied. There exist several competing constant multiplication techniques, with different relevance domains: they are reviewed in Chap. 12.
- Division by a constant can be implemented by multiplication with the inverse or by a specialization of the paper-and-pencil algorithm. This will be reviewed in Chap. 13.
- Constant addition can also be optimized, and incrementation (the addition of 1 to an integer) is a particularly common and useful operation. This is, however, a relatively simple logic optimization problem, and it does not deserve to be discussed further. Still, Chap. 5 will address more interesting variations of this problem, such as the joint computation of $X + Y$ and $X + Y + 1$.

It is also possible to specialize an operator thanks to more subtle relationships between its inputs. Here are two examples which will be expanded in later chapters:

- If two inputs to a multiplier are identical, then this multiplier computes a square, an operation which is simpler than arbitrary multiplication. A squarer operator will be cheaper than a multiplier.
- If it is known that the two inputs to a floating-point addition always have the same sign, then this addition is cheaper to implement than a standard addition: the cancellation case (see Sect. 11.1 of this book) never happens, which saves one leading-zero counter and one shifter [LTM03]. This situation may also happen in the intermediate data of composite operators. For instance, in an Euclidean norm such as $r = \sqrt{x^2 + y^2}$, both squares are positive.

There are more general specialization opportunities. For instance, many functions can be optimized if their inputs are known (in the context of the application) to lie within a certain range. Here are two examples:

- If a floating-point number is known to lie in $[-\pi, \pi]$, its sine is much cheaper to evaluate than in the general case.
- In general, if the range of the input to an elementary function is small enough, a low-degree polynomial approximation may suffice.

Finally, an operator may have its accuracy degraded, as long as the requirements of the application are matched. A spectacular example is the truncated multiplier: allowing an accuracy loss of one bit may save almost half the area of a multiplier [SWS00; Ban+10] – this will be detailed in Chap. 8. In the context of application-specific arithmetic, the loss of accuracy can typically be recovered at a much lower cost by adding one bit to the subsequent datapath.

Figure 1.4 shows the dataflow pipeline of Fig. 1.2 with two of these optimizations applied. The multiplier computing the square has been replaced with a squarer. The multiplier that had as input the constant 4 has been replaced with a constant multiplier. In fixed-point, this is just a constant shift, i.e., wiring. In floating-point, it is an addition on the exponent field, which is very small (even for double-precision it has only 11 bits).

Note that none of these two optimizations is particularly application-specific: here the optimizations are deduced from a local context, independently of the requirements of the wider application. Still, Fig. 1.4 is using two non-standard, hence application-specific operators.

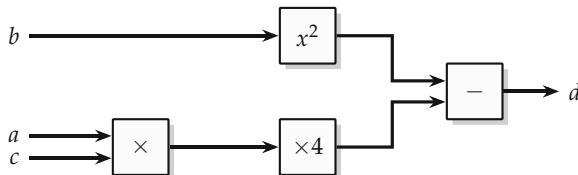


Fig. 1.4 Arithmetic optimizations on $d = b^2 - 4ac$.

1.4.2 Operator Fusion

It is sometimes possible to merge several operators into a single one in order to save intermediate computations. Take, for instance, the expression $\frac{x}{\sqrt{x^2 + y^2}}$. Considered as a black box, it has two inputs and one output, just like addition. Operator fusion consists in implementing this black box directly, instead of assembling two multipliers, an adder, a square root and a divider. Chapter 7 will discuss *bit heaps*, a very generic technique for the fusion of fixed-point sums of products (e.g., $x^2 + y^2$ in the previous example) in a single operator. This technique will be applied throughout the book inside coarser operators.

Operator fusion is also particularly relevant for floating-point operations. There, each individual operation involves normalization and rounding. From the black box point of view, however, what matters is that the end result is normalized and rounded [Lan08].

Operator fusion can go beyond the elimination of redundancies. In particular, for algebraic compositions such as $\frac{x}{\sqrt{x^2 + y^2}}$, it is often possible to directly derive a specific hardware algorithm. Indeed, an expression is said to be algebraic if it describes the roots of a (possibly multivariate) polynomial equation. This enables a very general family of methods to evaluate such expressions: somehow guess a poor approximation to the solution then refine it against the polynomial equation (which by definition will use only additions and multiplications). This is how we compute divisions and square root by hand or by machine [EL04], and it has been used for 2D and 3D Euclidean norms [TK00] and many other functions [Erc77]. This technique will be used in several chapters of this book.

There are also more subtle benefits to operator fusion. One is that accuracy can be improved with respect to a composition of individual operators. Another one is that more parallelism can be extracted. To illustrate all this, let us consider the floating-point implementation of the sum of squares $r = x^2 + y^2 + z^2$ (as required in the 3D Euclidean norm). Two optimizations seen in the previous section apply:

- use squarers instead of multipliers,
- use simplified floating-point adders, since all the squares are positive.

Operator fusion also brings the following benefits:

- In floating point, the significands of x^2 , y^2 and z^2 must be aligned before addition. In a fused operator, this may be done in parallel, reducing the latency or pipeline depth.
- With an operator built by assembling floating-point adders and multipliers, there is a total of 5 rounding errors involved. This entails that $\lceil \log_2(5) \rceil = 3$ bits of the result may be wrong. A fused operator may be designed to be more accurate than this.
- More subtly, the assembly of classic floating-point operators entails an implicit parenthesization such as $(x^2 + y^2) + z^2$. However, floating-point addition is not associative. There will be some rare cases when the value of the result will change when one exchanges x and z , due to the order of the two consecutive roundings. Such asymmetries may lead to very subtle bugs, such as a (correct) sorting algorithm going into an infinite loop [Mul+18]. A solution to this is to implement a fused operator whose architecture is fully symmetrical in its three inputs.

Such an architecture is depicted in Fig. 1.5 for a floating-point format with w_F bits of the significand and w_E bits of the exponent. It computes the three squares in parallel, truncates them to $w_F + g$ bits, and then aligns them to the largest one and adds them. The parameter g is a number of extra bits (“guard bits”) added to the internal datapath to control its accuracy. It can be shown that $g = 4$ ensures that this architecture is much more accurate than the composition of classic floating-point operators. Synthesis of the resulting

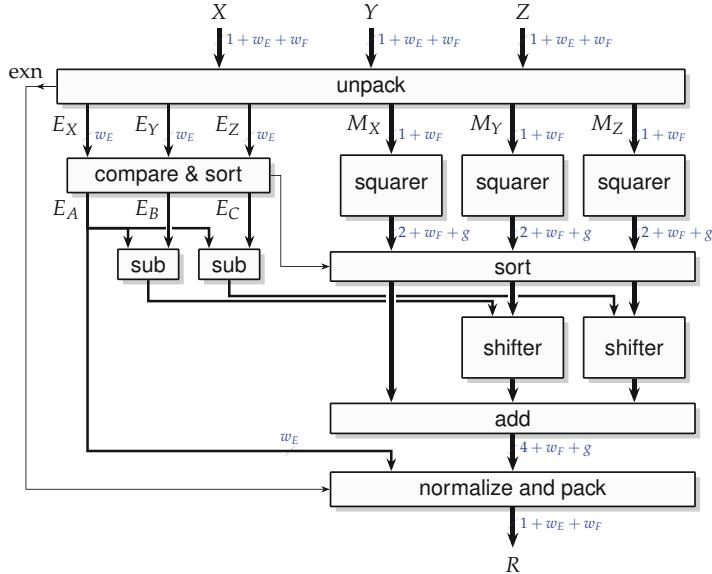


Fig. 1.5 An application-specific operator for the floating-point 3D squared norm.

fused operator on FPGA, for various floating-point formats, compared to a combination of operators working at the same frequency, shows that:

- latency is divided by 3,
- logic resource consumption is divided by 2 to 3,

for an operator that is functionally better, since it is more accurate and symmetrical in its inputs [DP11].

1.4.3 Function Approximation

Basic operator fusion techniques as used in the previous example essentially keep the arithmetic expression unchanged. Only the implementation of its operators or sub-components are optimized for their specific context. *Approximation*, conversely, replaces a mathematical expression with a completely different one that will hopefully compute a result of comparable numerical quality (or at least of sufficient quality for the application at hand).

Approximation can be used for algebraic functions (see, for instance, [PB02; WBL06; Din+10; Pas12; IP17] and the references therein for $1/x$, x/y , $1/\sqrt{x}$ and \sqrt{x}). It is unavoidable for transcendental functions. In this case we no longer speak of *computing* the function, but of *evaluating* it – although it makes little difference from a practical point of view.

The prevalent technique is polynomial approximation, which replaces a function $f(X)$ with a polynomial $p(X) = C_0 + C_1X + C_2X^2 + \dots + C_dX^d$ of degree d . Thus, the evaluation of the function reduces to multiplications and additions. This only works on a limited interval and under certain conditions (for instance, the function must have continuous derivatives on this interval up to some order). When it works, polynomial approximation involves a multi-faceted trade-off: Accuracy is better with smaller intervals and/or higher degrees. Costs are higher with higher degrees.

Considering this trade-off, *range reduction* consists in transforming the problem, for instance, by a change of variable, into a simpler one (i.e., one that will be cheaper to evaluate). As the name implies, the transformed problem usually requires the evaluation of a function on a smaller interval.

Some range reduction techniques are function-specific, for example, trigonometric identities can be used to reduce the range of trigonometric functions to one eighth of a period. Some techniques are generic. For example, Fig. 1.6 shows an architecture exploiting a *uniform piecewise approximation* scheme. Here, the function input X is split into its leading bits A and its lower bits Y . This splits the input domain into smaller subdomains, indexed by A . On each subdomain, a polynomial $p_A(Y)$ approximates the function (its degree is $d = 3$ in Fig. 1.6). It is evaluated using the Horner scheme: $p_A(Y) = C_0 + Y \times (C_1 + Y \times (C_2 + Y \times C_3))$, where the coefficients C_i have been precomputed and stored in a table addressed by A . This technique, and others, will be studied in detail in Part III.

Many range reduction techniques (including the one depicted in Fig. 1.6) are based on tabulating polynomial coefficients or other pre-computed values. Therefore, they add *memory consumption* to the implementation trade-off.

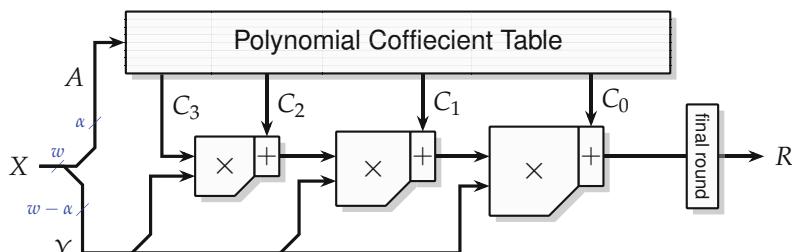


Fig. 1.6 A fixed-point polynomial evaluator, using uniform segmentation and a Horner scheme with truncated multipliers – see Chap. 18 for more details.

Note that the simplest option is to precompute (in software) all the possible values of the function and store them in a read-only memory. The memory consumption is exponential with respect to the input size (in bits), so this option does not scale well. However, it is a very generic and useful basic building block for more complex techniques.

Function approximation techniques have been extensively studied for functions of one variable and are the subject of part III of this book. Part IV will show how to rely on these techniques to implement some useful elementary functions in fixed point or floating point.

1.4.4 Resource Sharing

At the application level, rarely used computing blocks can be time-multiplexed between several computations. This is actually the way microprocessors share their arithmetic units. In the present book, we are more interested in sharing arithmetic components at a finer scale between arithmetic units. Such sharing is not only performed by time multiplexing. A very powerful technique is to expose sub-computations that can be used several times within a larger computation (*sub-expression sharing*). Here are a few examples:

- The reason why a squarer is smaller than a full multiplier is that each digit-by-digit product appears twice: it can be computed only once, as Fig. 1.7 illustrates. More details can be found in Chap. 14.
- Still in multipliers, the Karatsuba technique reduces the number of small multipliers needed to implement a larger one by sharing subproducts. This technique will be presented in detail in Chap. 8.
- We have already mentioned that multiplication by a constant can be computed by a sequence of additions. To minimize the number of such additions, it is possible to try and organize them to maximize the re-use of the intermediate values computed by these additions. For instance, the value 2193 can be written as $2193 = 17 + (17 \ll 7)$, where \ll denotes the binary left shift operator (C-like syntax). Therefore, the multiplication of this constant by an integer X can be written $2193X = 17X + (17X \ll 7)$: the value $17X = (X \ll 4) + X$ can be computed once and reused twice. Chapter 12 will explore this technique systematically. Such sharing can also be found in the related problems of multiple constant multiplication and constant matrix multiplication, which will also be studied in this chapter.
- Bit-serial arithmetic re-uses the same full adder cell to compute each bit of an addition. In some situations, this improves computation throughput [LS17].

Resource sharing will not be treated in a specific chapter, but it will be a pervasive concern in the search for arithmetic efficiency.

$\begin{array}{r} 2321 \\ \times \quad 2321 \\ \hline \end{array}$	<p>Left: all the digit-by-digit products below the diagonal line have already been computed above it.</p> <p>Right: the same result is obtained by counting twice each digit in bold.</p> <p>In binary, multiplication by 2 will resume to rewiring.</p>	$\begin{array}{r} 2321 \\ \times \quad 2321 \\ \hline \end{array}$
		$\begin{array}{r} 2321 \\ 464 \\ 69 \\ \hline 5387041 \end{array}$

Fig. 1.7 Paper-and-pencil algorithm for squaring a decimal number.

1.4.5 Target-Specific Optimizations

In this book, we consider two broad families of targets: on one side ASICs and on the other side FPGAs from a variety of vendors. Designing for a technology target may seem less interesting in a textbook. However, it may bring considerable improvements in performance and resources.

The main message here is that the technology does not only determine the value of some parameter of an algorithm. It may determine the algorithm itself. Here are a few examples:

- The logic in FPGAs is built around the small look-up table (LUT). This has led to specific algorithms for constant multiplication [Cha94] which will be reviewed in Chap. 12.
- In ASICs, we have a choice of adders, from low-area but slow ones to fast but large ones. The cheapest 32-bit adder will at least have the delay of 32 basic gates. Conversely, in most FPGAs, there is some form of built-in addition support in the form of fast carry chains, where “fast” means “faster than the usual, programmable routing” (more details will be found in Chap. 4). The effect is that a 32-bit adder will have a delay comparable to one basic gate. In other words, cheap adders are comparatively faster on FPGAs than on ASICs – the real reason being, of course, that other logic functions are comparatively slower due to the programmable routing.
- The LUTs in FPGAs are usually located in front of the fast carry chains (see Chap. 5). These can be used to expand the functionality of the adder. Examples are the ternary (3-input) adders described in Chap. 5 or elements found in compressor trees as described in Chap. 7.
- Some FPGAs also have embedded multiplier blocks and small memories. The size and quantity of these objects in an FPGA is fixed by the vendor,

and the objective of the application designer is to make the best use of it. There is much more freedom when designing ASICs. Actually, many parameters that are degrees of freedom when designing for ASIC targets become constraints when designing for FPGAs. This does not deeply change the design space, but it deeply changes its exploration.

For similar reasons, many classic arithmetic techniques, such as compressor trees (reviewed in Chap. 7) or Karatsuba multiplication (reviewed in Chap. 8), have been revisited in the context of FPGAs.

1.5 General Design Principles for Application-Specific Arithmetic

As the previous section has shown, designing application-specific arithmetic is more challenging than designing traditional arithmetic: we have more possibilities to design operators, and they are more parameterized. We now expose some general design principles that will help designers tackle this challenge. These principles also differentiate the present book from other arithmetic books.

1.5.1 Parameterize

In application-specific arithmetic, any operator should be as parametric as possible, with parameters such as external and internal word sizes, order of a polynomial, etc. It does involve more work to design a fully parametric operator than to write a single instance, but it has several advantages.

- It is future proof: a parametric operator will be easier to adapt to a later technology, to a change of the algorithm. It can be more easily reused in a different design.
- It enables exhaustive testing of small-scale operators. This will uncover corner-case bugs that random testing of the full-scale operator would likely miss. For instance, exhaustive testing of a 2-input binary 64-bit floating-point operator requires $2^{2 \times 64} \approx 3.4 \cdot 10^{38}$ tests, which is unfeasible. However, exhaustively testing the 16-bit version requires $2^{32} \approx 4 \cdot 10^9$ tests, which is perfectly achievable on current computers. If the code is the same as for the 32-bit operator, this gives much more confidence in the latter. However, one should of course be aware that corner-case bugs may depend on size parameters.
- It eases optimization. There are often design parameters for which the optimal value is not obvious. Instead of committing to a (more or less random) choice for such parameters, it is better to keep them open. This will

enable design exploration and, in many cases, the formalization of an optimization problem to determine the best value for a given context.

- In general, writing parameterized code gives a much deeper understanding of the implementation and much more confidence in it.

Parameters of interest broadly fall in two classes.

Functional parameters determine the functionality. They include:

- data type (fixed-point or floating-point format, signed versus unsigned fixed-point inputs, for instance),
- data size (bit-width),
- operation parameters (for instance, the constant in a constant multiplication).

Performance parameters do not determine any functionality but have an effect on performance (and often on the resources used). They include:

- implementation options (pipelined versus iterative, algorithm to be used, internal parameters),
- target technology (which FPGA, which ASIC technology),
- target performance constraints.

1.5.2 Compute Just Right (*Last-Bit Accuracy*)

Common sense tells us that for efficiency, a hardware implementation should not compute bits which hold no useful information.

Consider, for instance, the exponential function. On a 32-bit processor that imposes a data granularity of 32-bits, an implementation of this function must have 32-bit input and output. The standard in this case is the (32-bit) single-precision floating-point format. This format has a 23-bit significand; therefore no single-precision exponential implementation may offer a relative accuracy better than 2^{-24} . Usually the implementation in the mathematical library (libm) offers an accuracy that is close to this limit.

However, some applications will be contented with a much worse accuracy, say 2^{-8} . If this saves resources or time, it makes sense to design such a degraded software implementation. Then almost half of the 32 output bits will contain noise instead of useful information. However, removing them from the processor datapath is not an option.

When designing for ASIC or FPGAs, it is an option, as data formats are much more flexible. Here, using a single precision exponential core accurate to 2^{-8} would mean that, out of the 32 bits passed along the datapath, 16 bits hold useless noise. Obviously, this would entail a waste of precious routing resources and registers. Besides, the subsequent operators in the datapath

would compute on this noise, meaning more wasted resources and useless power consumption.

A much better option is to use an implementation of the exponential whose output has an 8-bit significand (and is accurate to the last of these 8 bits). Not only will it be smaller for the same accuracy; it also allows the circuit around it to be smaller as well.

More generally, we claim that no operator should be designed that is not accurate to its last bit. If an application, at some point, requires a relative accuracy of 2^{-8} , then the floating-point format it uses at this point should have only 8 bits of significand. All the input bits should be taken into account, and all the output bits should hold relevant information.

One ambition of this book is to help designers tackle this *arithmetic efficiency* challenge.

Designing arithmetic-efficient datapaths may take considerable effort. It requires solving two sub-problems: (1) defining which accuracy is required at each point of a datapath and (2) using, at each point, operators that compute just right for this accuracy.

This book directly addresses the second, easier, sub-problem: the design of operators with parameterized input/output formats that are as accurate as their output format allows, at the minimal cost. The deep relationship between format and accuracy, already evoked in Sect. 1.3, is called rounding and will be studied in detail in Chaps. 2 and 3.

A consequence with respect to our first design principle (parameterize) is that there is generally no need for an “accuracy” parameter. Indeed, we must already define output format parameters. In an arithmetic-efficient design, these format parameters also define the accuracy of the circuit.

The first sub-problem (which accuracy is required at each point of a datapath) is the more difficult, in particular because it is deeply application-dependent: it first requires to understand the accuracy requirements of the application, understand the accuracy of its inputs, and then propagate this information inside the datapath. There is no general technique or framework for this that will work for all applications.

However, this book provides several examples of arithmetic-efficient designs, for coarse arithmetic operators. For instance, consider again the polynomial evaluator of Fig. 1.6. There, all the wires have different bit-widths, and these bit-widths may be computed by a mathematical analysis that takes into account the function to approximate, the input interval, and the output precision. In this case, the first sub-problem, defining which accuracy is required at each point of a datapath, can be solved thanks to the very mathematical nature of the computation to implement. The last chapters of this book provide even coarser examples.

1.5.3 Expose the Design Space

This principle is firstly a rephrasing of the need for heavily parameterized operators: it is important to expose all the parameters of the design space.

However, there is another important aspect: whenever possible, one should design, along with parameterized operators, models of their cost and performance (using whatever relevant metrics). Such models should be simple enough to enable design-space exploration without having to perform costly syntheses. They should be accurate enough to be able to predict the result of such syntheses in a useful way, but no more: synthesis tools are capable of increasingly aggressive optimizations, whose result is very difficult to predict. Note that for the purpose of design-space exploration, the monotonicity of the models is often more important than their accuracy.

1.5.4 Do Not Write Operators, Write Generators

For all the reasons given so far, it makes little sense to design a *library* of application-specific operators directly in an hardware description language (HDL). The state of the art is to design *generators* of hardware descriptions. These generators input some or all of the operator parameters and output the HDL description of the operator. This is illustrated by Fig. 1.8.

Historically, such generators were first written because the number of parameter combinations was too large for what is practical in hardware description languages. However, the generator approach has many other advantages:

- Some of these parameters are not simply numbers. For instance, back to the polynomial approximation architecture of Fig. 1.6, one of its parameters is the mathematical function it approximates. In this book, we will use it for $f(x) = \sin(\frac{\pi}{4}x)$ and for $f(x) = e^x - 1 - x$ among others. It is important that a designer may provide an arbitrary mathematical function as this parameter, as illustrated by Fig. 1.8.
- The generator approach opens the way for complex optimization techniques well beyond the reach of mainstream HDLs such as VHDL or Verilog. For instance, in Chap. 12, a constant multiplication architecture is derived from the constant thanks to the solution of an integer linear programming (ILP) problem.

In practice, an operator generator is a program that can be interfaced to other programs and libraries: one for parsing arbitrary function specifications, one for solving ILP problems, etc. This program inputs the user-provided parameters, performs design-space exploration, and then outputs the HDL code for the selected solution. It is important to understand that

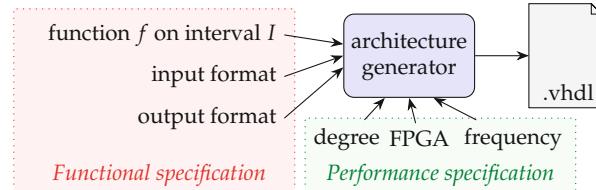


Fig. 1.8 An example of generator: a generic fixed-point polynomial approximator that will produce the architecture of Fig. 1.6.

the user does not have to input all the internal parameters (such as the bit-widths in Fig. 1.6): these parameters are computed out of the user-specified parameters. It is a general design principle that the user should input only a few high-level parameters and leave as much as possible to the tools.

1.5.5 Generate the Test Bench Along with the Operator

Application-specific arithmetic addresses an infinite number of operators, and there is no way they can all be verified. Formal proof techniques have been used on single operator instances, but formally proving an arithmetic generator is currently beyond the capabilities of formal proof tools.

Fortunately, it is possible to generate a test bench for an instance of a parameterized operator. Actually it is much easier than to generate the operator itself. The reason is that in most cases, arithmetic operators are, mathematically speaking, simple objects. Even if the internal construction of a floating-point sum of squares is complex, its mathematical specification is very concise: compute $x^2 + y^2 + z^2$, and return a value that obeys some rounding specification.

In practice, one can rely on arbitrary precision libraries such as GMP and MPFR [Fou+07] to implement these mathematics. MPFR, for instance, offers support for a wide range of elementary functions. With minimal generator-wide support for data-type conversions, a parametric test bench generator can be written in very few lines of code.

Another advantage of testing an operator against its mathematical specification is that the implementation of the test program is very unlikely to be subject to the same bugs as the implementation of the operator itself.

1.6 Organization of This Book

The present book attempts to review the state of the art of application-specific arithmetic. An overview of its chapters is given in Fig. 1.9. The book

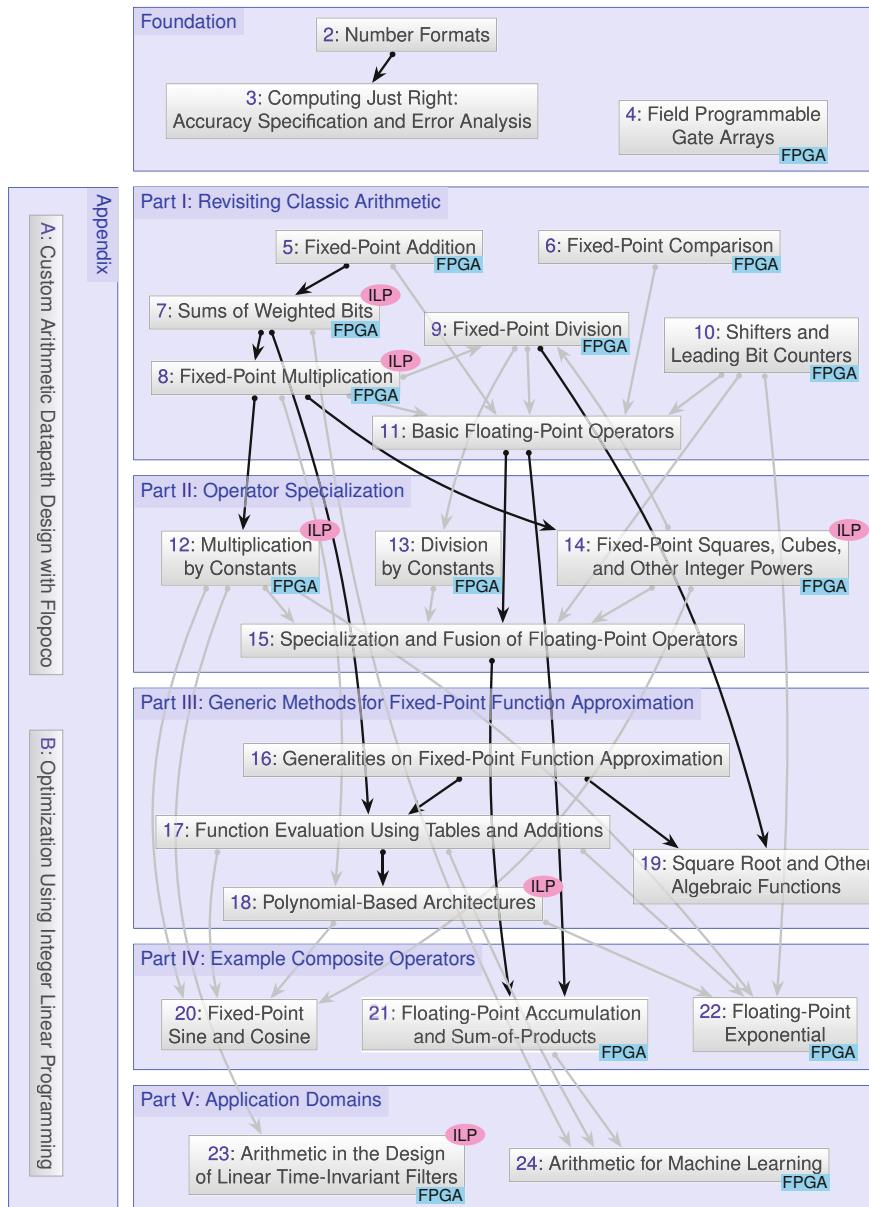


Fig. 1.9 Dependencies between the chapters of this book.

is structured in four main parts (the light blue boxes), preceded by three foundation chapters, and followed by two appendix chapters.

The book can be read linearly. For readers wishing to escape this order, each arrow in Fig. 1.9 indicates when a chapter uses some knowledge from another chapter. Black arrows indicate prerequisite reads, and lighter arrows indicate that a chapter involves components described in another chapter. For instance, a floating-point exponential, described in Chap. 22, can be constructed by assembling multipliers by constants, tables, function approximators, and shifters. However, Chap. 22 can be read independently of the chapters describing these components, provided the reader is ready to consider them as black boxes.

Chapters 2 and 3 build the foundation of this book, so most subsequent chapters depend on them. Also, many chapter include optimizations specific to FPGAs, for which the foundations are presented in Chap. 4. Readers not interested in FPGA arithmetic may just skip Chap. 4 and the FPGA-specific sections.

Some optimization problems are solved in this book using integer linear programming (ILP). The reader unfamiliar with ILP will find in Appendix B an introduction to this optimization tool.

The other appendix chapter describes the FloPoCo software which we briefly introduce now.

1.7 Support Software: The FloPoCo Project

The systematic study of application-specific arithmetic is also the object of the FloPoCo project (<http://www.flopoco.org/>). FloPoCo attempts to implement the design principles exposed above and offers open-source implementations of most of the operators presented in this book. It is therefore a good way for the interested reader to explore in more depth the opportunities of application-specific arithmetic. Examples using FloPoCo are provided throughout the book, and more details on the internals of FloPoCo are given in Appendix A.

1.8 Other Books on Computer Arithmetic

Here are a few other books that will be of interest to the readers of the present book.

1.8.1 General Computer Arithmetic

- Miloš D. Ercegovac and Tomás Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004 [EL04]
This reference textbook covers number representations and the basic operations, in integer or floating point. It is especially strong on division and square root and also includes algorithms for exponential and logarithm, as well as the CORDIC function evaluation technique. This textbook also provides an in-depth coverage of digit-serial arithmetic techniques that have (for this reason) been left out of the present book.
- Behrooz Parhami. *Computer Arithmetic, Algorithms and Hardware Designs*. 2nd ed. Oxford University Press, 2010 [Par10]
This is another comprehensive textbook on computer arithmetic covering number systems, basic operations, function evaluation (including CORDIC), and modular arithmetic. It also addresses implementation issues, from low power and fault tolerance to reconfigurable computing.
- Israel Koren. *Computer Arithmetic Algorithms*. 2nd ed. Prentice-Hall, 2002 [Kor02]
This book is another comprehensive review of computer arithmetic, from number systems to hardware implementation. It also covers a selection of elementary functions, as well as two exotic number systems that we had to leave out of the present book: the Residue Number System (RNS) and the Logarithm Number System (LNS).
- Michael J. Flynn and Stuart F. Oberman. *Advanced Computer Arithmetic Design*. Wiley-Interscience, 2001 [FO01]
This book presents advanced methods of computer arithmetic in VLSI with a focus on the elementary operations of addition, multiplication, and division but also on specialized techniques like wave pipelining.
- Donald Knuth. *The Art of Computer Programming, vol. 2: Seminumerical Algorithms*. 3rd ed. Addison Wesley, 1997 [Knu97]
This is the volume of this series that deals with computer arithmetic. The focus is on software, but many useful core concepts and algorithms are well explained there.

1.8.2 Arithmetic for FPGAs

- Jean-Pierre Deschamps, Gustavo D. Sutter, and Enrique Cantó. *Guide to FPGA Implementation of Arithmetic Functions*. Vol. 149. Lecture Notes in Electrical Engineering. Springer, 2012 [DSC12]
This book focuses on the implementation of computer arithmetic circuits on FPGAs. After introducing the aspects of digital circuits, different implementations are given for basic operators and special functions.

- Ayan Palchaudhuri and Rajat Subhra Chakraborty. *High Performance Integer Arithmetic Circuit Design on FPGA*. Vol. 51. Springer Series in Advanced Microelectronics. 2016 [PC16]
This book is another one focusing on the implementation aspects of integer computer arithmetic circuits on FPGAs. It covers the basic integer operations but is more specialized on low-level and vendor-specific optimizations.
- Uwe Meyer-Baese. *Digital Signal Processing with Field Programmable Gate Arrays*. 4th ed. Springer, 2014 [Mey14]
The book covers the implementation of digital signal processing on FPGAs. It contains a section on basic computer arithmetic followed by a detailed discussion on how to implement specific algorithms from digital signal processing. It can serve as a background for applications, e.g., digital filters or the Fourier transform, which can benefit from the operator specialization discussed in Part II of the present book.

1.8.3 Other Specialized Books

- Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldeş, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. 2nd ed. Birkhäuser Boston, 2018 [Mul+18]
This book covers all aspects of floating-point computing with a strong focus on the IEEE 754-2008 standard for floating-point arithmetic. It includes chapters on the construction of standard floating-point units. It complements Chap. 11 (among others) of the present book.
- Jean-Michel Muller. *Elementary Functions, Algorithms and Implementation*. 3rd ed. Birkhäuser Boston, 2016 [Mul16]
This book addresses the evaluation of elementary functions in hardware and software. It complements Part III of the present book, or maybe the other way around.
- Amos Omondi. *Computer-Hardware Evaluation of Mathematical Functions*. Imperial College Press, 2016 [Omo16]
This book addresses the evaluation of elementary functions in hardware, as the title suggests, but also high-speed arithmetic for the basic operations. Concerning elementary functions, it is nearly exhaustive in terms of scope but remains at a very mathematical level. The present book complements it by being more focused and detailed down to hardware considerations.
- George A. Constantinides, Peter Y.K. Cheung, and Wayne Luk. *Synthesis and optimization of DSP algorithms*. Kluwer, 2004 [CCL04]
This book is focused on digital signal processing algorithms on FPGAs. It addresses several topics that are primarily arithmetic, in particular word-length optimization, and saturation arithmetic.

- Ulrich W. Kulisch. *Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units*. Springer-Verlag, 2002 [Kul02]
This book explores ways of improving the arithmetic of general-purpose processors: exact floating-point dot-products and interval arithmetic support.
- Peter Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000 [Mar00]
This book is essentially software-oriented but presents a good panorama of evaluation techniques for the most important elementary functions in floating point.

1.8.4 Approximate Computing

The present book fits in a more general trend in the recent literature called “approximate computing.” This phrase encompasses many techniques that deliberately degrade the accuracy of some computation, as long as the final result is good enough. This is strongly application-specific and exactly the purpose of the “computing just right” approach. However, we tend to avoid the phrase “approximate computing” for two main reasons. The first is that any computation in finite precision (i.e., any computation by a machine) is by definition approximate. The second is that our purpose is to capture these inevitable approximations very accurately.

Some of the approximate computing literature studies application-level accuracy, for instance, trying to minimize the size (in bits) of internal signals, a problem called Word Length Optimization (WLO) [CCL04]. The present book complements such approaches since it systematically addresses non-standard word lengths. However, it does not address the WLO problem, except in the very specific case where the application under consideration is itself a larger arithmetic operator (indeed Chaps. 20, 22, and 23 include instances of the WLO problem).

Approximate computing also includes arithmetic design approaches which are not covered in the present book, in particular adders or multipliers that are accurate for most inputs, but sometimes very inaccurate (due, for instance, to simplified logical equations or to extreme voltage down-scaling which reduces the power consumption of operators, but makes their execution randomly faulty). We prefer in this book to focus on operators whose accuracy is consistent and therefore do not require any assumption on the statistical distribution of the input data. One reason is that such arithmetic is more easily composable (and the book will give several examples of such compositions). Another reason is that energy is dissipated in data transfers more than in computation, and therefore it seems better to make the most out of each bit.

Finally, approximate computing also addresses many techniques which are not related to arithmetic, such as, for instance, the approximation of the control flow of a program. Two recent books together provide a comprehensive review of its many aspects.

- Alberto Bosio, Daniel Ménard, and Olivier Sentieys, eds. *Approximate Computing Techniques: From Component- to Application-Level*. Springer, 2022 [BMS22]

This book is a survey of approximate computing covering arithmetic and hardware, language and compiler level, and applications.

- Weiqiang Liu and Fabrizio Lombardi, eds. *Approximate Computing Techniques*. Springer, 2022 [LL22]

Compared to the previous one, this book is focused on hardware but covers more of it, for instance, approximate computing for finite-field arithmetic and cryptography.

1.9 Notations Used in This Book

This section is intended for reference and can be skipped at the first read.

Here are a few conventions about the notations used in this book (unless explicitly defined otherwise); Table 1.1 summarizes these common notations:

- The symbols \mathbb{R} , \mathbb{Z} , and \mathbb{N} are used for real, integer, and natural numbers (including zero), respectively.
- The symbol \mathbb{F} denotes a finite set of real numbers or special values that can be represented in some finite-precision format. This symbol is used in contexts where the format itself (which can be floating-point, fixed-point, machine integers, or other) is irrelevant. If the format fits on w bits, the cardinal of \mathbb{F} is at most 2^w values. It may be smaller than 2^w if the format is redundant.
- Words (vectors of bits) are denoted using capital letters (e.g., X).
- A single bit of a word X is usually denoted x_i . Wherever possible, the index i is given to the bit of weight 2^i in the word. For instance, if X is a fixed-point number with two integer bits and two fractional bits, its bit representation can be written $x_1x_0.x_{-1}x_{-2}$.
- A sub-word of a word X is denoted by specifying the range in square brackets, e.g., $X[5\dots 3]$ means the sub-word consisting of the bits x_5 , x_4 and x_3 . Here x_5 is the most significant bit (MSB), and x_3 is the least significant bit (LSB).
- The word length (in bits) of a word X (sometimes called *word size*, or *bit-width*, or simply *width* of X) is denoted by w_X . In general w denotes a word length.
- In general a tilded variable \tilde{X} is less accurate than the same variable without tilde X .

- The Greek letter δ usually describes an absolute error, which is the difference between a less accurate value \tilde{A} a more accurate value A :

$$\delta = \tilde{A} - A.$$

- The Greek letter ε usually describes a relative error, i.e., the relative difference between a less accurate value \tilde{A} a more accurate value A :

$$\varepsilon = \frac{\tilde{A} - A}{A}.$$

- We tend to use Greek letters for other integer parameters. In particular, a high-radix binary representation (introduced in detail in Sect. 2.3) groups bits in chunks of α bits, and the radix is denoted as β . For instance, hexadecimal is $\beta = 16$ for $\alpha = 4$.
- A signed fixed-point format is denoted by $\text{sfix}(m, \ell)$ and an unsigned fixed-point format by $\text{ufix}(m, \ell)$. The two integers m and ℓ denote, respectively, the positions of the MSB and LSB of the format. For instance, the format of signed fixed-point data in $[-1, 1]$ on w bits will be $\text{sfix}(0, -w + 1)$: the sign bit has weight 2^0 and the LSB has weight 2^{-w+1} . The sign bit of a two's complement signed number X will be denoted as s_X or s when there is no ambiguity. More details on fixed-point formats are given in Sect. 2.2.2.
- For fixed-point formats,
 - Truncating a finite precision binary number X to a lower precision, i.e., with a larger LSB position ℓ , will be denoted as $\lfloor X \rfloor_\ell$. Formally $\lfloor X \rfloor_\ell = \max\{Y \text{ s.t. } Y \leq X, 2^{-\ell}Y \in \mathbb{N}\}$, but from a hardware point of view, truncation consists in dropping all the bits to the right of position ℓ .
 - Rounding a finite precision binary number X to a lower precision, i.e., with a larger LSB position ℓ , will be denoted as $\lceil X \rceil_\ell$. Mid-points (tie cases that could be rounded up or down) are rounded up, to match the typical hardware implementation $\lceil X \rceil_\ell = \left\lceil X + 2^{\ell-1} \right\rceil_\ell$.
 - An unspecified quantization of a finite precision binary number X to a fixed-point format with LSB ℓ will be denoted as $[X]_\ell$. Such a quantization may be rounding, truncation, or another function, as long as the corresponding error $[X]_\ell - X$ can be bounded by $c \cdot 2^\ell$ for some small constant c .
- In a floating-point number X , the sign bit is denoted as s_X or s , the exponent field is denoted as E_X or E , and the mantissa fraction field is denoted as F_X or F . More details on floating-point formats are given in Sect. 2.5.
- Boolean expressions use the symbols \wedge , \vee , and \oplus to express the AND, OR, and XOR operations, respectively. Also, \bar{x} denotes the Boolean negation. For the sake of readability, we may omit the \wedge operator in expressions like $x \wedge y = xy$.

- For denoting intervals, we use square and round brackets to indicate if the value is included or not, respectively, e.g., a number $X \in (1, 2]$ is such that $1 < X \leq 2$.

A list of common acronyms is available at the end of this book.

Table 1.1 Common notations used throughout the book.

Symbol	Meaning
\mathbb{R}	Set of real numbers
\mathbb{Z}	Set of integer numbers (including zero)
\mathbb{N}	Set of natural numbers (including zero)
\mathbb{F}	Set of machine numbers
x_i	The i -th bit of X (having weight 2^i)
$X[b \dots a]$	A subword of X consisting of bits at positions b to a (included)
w	A word size in bits
w_X	The word size of signal X in bits
α	A small number of bits
β	The radix of a number system
\tilde{X}	A less accurate value of X
δ	An absolute error
$\bar{\delta}$	An upper bound on the absolute value $ \delta $ of δ
ε	A relative error
m	The MSB of a fixed-point format
ℓ	The LSB of a fixed-point format
u	The value of the unit in the last place (ulp) $u = 2^\ell$
$\text{sfix}(m, \ell)$	A signed fixed-point format with bits at positions ℓ to m (included)
$\text{ufix}(m, \ell)$	An unsigned fixed-point format
$\lfloor X \rfloor$	Truncation of X to the largest integer smaller than or equal to X
$\lfloor X \rfloor_\ell$	Truncation of X to a fixed-point format of LSB position ℓ
$\lfloor X \rfloor_{\text{r}}$	Rounding of X to the nearest integer
$\lfloor X \rfloor_\ell$	Rounding of X to a fixed-point format of LSB position ℓ
E or E_X	The exponent of a floating-point number X
F or F_X	The mantissa fraction of a floating-point number X
s or s_X	The sign bit of a floating-point number X
\wedge	Logical AND relation
\vee	Logical OR relation
\oplus	Logical exclusive-or (XOR) relation
\mathbf{x}	A vector \mathbf{x}
\mathbf{X}	A matrix \mathbf{X}

References

- [754-19] *IEEE Standard for Floating-Point Arithmetic*. also IEEE/ISO/IEC 60559-2020. 2019 (cit. on pp. 7, 330, 332, 357).
- [Ban+10] Sebastian Banescu, Florent de Dinechin, Bogdan Pasca, and Radu Tudoran. “Multipliers for floating-point double precision and beyond on FPGAs”. In: *ACM SIGARCH Computer Architecture News* 38 (2010), pp. 73–79 (cit. on pp. 10, 234, 238, 239).
- [BMS22] Alberto Bosio, Daniel Ménard, and Olivier Senteys, eds. *Approximate Computing Techniques: From Component-to Application-Level*. Springer, 2022 (cit. on p. 26).
- [Bru18] Javier D. Bruguera. “Radix-64 Floating-Point Divider”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2018, pp. 87–94 (cit. on pp. 4, 288, 349, 350).
- [Bru22] Javier D. Bruguera. “Low-Latency and High-Bandwidth Pipelined Radix-64 Division and Square Root Unit”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2022 (cit. on pp. 4, 353).
- [CCL04] George A. Constantinides, Peter Y.K. Cheung, and Wayne Luk. *Synthesis and optimization of DSP algorithms*. Kluwer, 2004 (cit. on pp. 24, 25).
- [Cer90] Paul E. Ceruzzi. In: *Computing before computers*. Iowa State University Press, 1990. Chap. 6: Relay calculators (cit. on p. 1).
- [Cha94] Ken D. Chapman. “Fast Integer Multipliers Fit in FPGAs”. In: *Electronic Design News* (1994) (cit. on pp. 15, 771).
- [CHT02] Marius Cornea, John Harrison, and Ping Tak Peter Tang. *Scientific Computing on Itanium®-Based Systems*. Intel Press, 2002 (cit. on pp. 2, 294, 347).
- [Din+10] Florent de Dinechin, Mioara Joldes, Bogdan Pasca, and Guillaume Revy. “Multiplicative square root algorithms for FPGAs”. In: *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2010, pp. 574–577 (cit. on pp. 13, 303, 307, 356).
- [DP11] Florent de Dinechin and Bogdan Pasca. “Designing Custom Arithmetic Data Paths with FloPoCo”. In: *IEEE Design & Test of Computers* 28.4 (2011), pp. 18–27 (cit. on p. 12).
- [DSC12] Jean-Pierre Deschamps, Gustavo D. Sutter, and Enrique Cantó. *Guide to FPGA Implementation of Arithmetic Functions*. Vol. 149. Lecture Notes in Electrical Engineering. Springer, 2012 (cit. on p. 23).
- [EL04] Miloš D. Ercegovac and Tomás Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004 (cit. on pp. 3, 11, 23, 214, 218, 259, 267, 303).
- [EL94] Miloš D. Ercegovac and Tomás Lang. *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, Boston, 1994 (cit. on pp. 3, 259, 300, 303).

- [Erc77] Miloš D. Ercegovac. "A General Hardware-Oriented Method for Evaluation of Functions and Computations in a Digital Computer". In: *IEEE Transactions on Computers* C-26.7 (1977), pp. 667–680 (cit. on pp. 11, 48).
- [FO01] Michael J. Flynn and Stuart F. Oberman. *Advanced Computer Arithmetic Design*. Wiley-Interscience, 2001 (cit. on p. 23).
- [Fou+07] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. "MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding". In: *ACM Transactions on Mathematical Software* 33.2 (2007), 13:1–13:15 (cit. on p. 20).
- [GB91] Schmuel Gal and Boris Bachelis. "An Accurate Elementary Mathematical Library for the IEEE Floating Point Standard". In: *ACM Transactions on Mathematical Software* 17.1 (1991), pp. 26–45 (cit. on p. 2).
- [IP17] Matei Istoan and Bogdan Pasca. "Flexible Fixed-Point Function Generation for FPGAs". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2017, pp. 123–130 (cit. on pp. 13, 260, 301).
- [Knu97] Donald Knuth. *The Art of Computer Programming, vol.2: Seminumerical Algorithms*. 3rd ed. Addison Wesley, 1997 (cit. on p. 23).
- [Kor02] Israel Koren. *Computer Arithmetic Algorithms*. 2nd ed. Prentice-Hall, 2002 (cit. on pp. 23, 211).
- [KR07] Ian Kuon and Jonathan Rose. "Measuring the Gap Between FPGAs and ASICs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2 (2007), pp. 203–215 (cit. on p. 5).
- [Kul02] Ulrich W. Kulisch. *Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units*. Springer-Verlag, 2002 (cit. on p. 25).
- [Lan08] Martin Langhammer. "Floating Point Datapath Synthesis for FPGAs". In: *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2008, pp. 355–360 (cit. on p. 10).
- [LL22] Weiqiang Liu and Fabrizio Lombardi, eds. *Approximate Computing Techniques*. Springer, 2022 (cit. on p. 26).
- [LS17] Aaron Landy and Greg Stitt. "Serial Arithmetic Strategies for Improving FPGA Throughput". In: *ACM Transactions on Embedded Computing Systems* 16.3 (2017) (cit. on p. 14).
- [LTM03] Jian Liang, Russel Tessier, and Oskar Mencer. "Floating Point Unit Generation and Evaluation for FPGAs". In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2003, pp. 185–194 (cit. on p. 9).
- [Mar00] Peter Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000 (cit. on pp. 4, 25, 260, 288, 347).
- [Mer09] R. Merrit. "ARM CTO: Power Surge Could Create 'Dark Silicon'". In: *EE Times* (2009) (cit. on p. 3).

- [Mey14] Uwe Meyer-Baese. *Digital Signal Processing with Field Programmable Gate Arrays*. 4th ed. Springer, 2014 (cit. on p. 24).
- [Mul+18] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldeş, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. 2nd ed. Birkhäuser Boston, 2018 (cit. on pp. 11, 24, 311, 314, 324, 330, 332, 336, 346, 347, 349).
- [Mul16] Jean-Michel Muller. *Elementary Functions, Algorithms and Implementation*. 3rd ed. Birkhäuser Boston, 2016 (cit. on pp. 24, 212).
- [OF97a] Stuart F. Oberman and Michael J. Flynn. “Design Issues in Division and Other Floating-Point Operations”. In: *IEEE Transactions on Computers* 46.2 (1997), pp. 154–161 (cit. on pp. 3, 4, 260).
- [Omo16] Amos Omondi. *Computer-Hardware Evaluation of Mathematical Functions*. Imperial College Press, 2016 (cit. on p. 24).
- [OS05] Stuart F. Oberman and Ming Y. Siu. “A High-Performance Area-Efficient Multifunction Interpolator”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2005 (cit. on p. 4).
- [Par10] Behrooz Parhami. *Computer Arithmetic, Algorithms and Hardware Designs*. 2nd ed. Oxford University Press, 2010 (cit. on pp. 23, 267, 287, 303).
- [Pas12] Bogdan Pasca. “Correctly Rounded Floating-Point Division For DSP-Enabled FPGAs”. In: *Field Programmable Logic and Applications*. 2012 (cit. on pp. 13, 260, 288, 291, 292, 303).
- [PB02] José-Alejandro Piñeiro and Javier D. Bruguera. “High-Speed Double-Precision Computation of Reciprocal, Division, Square Root, and Inverse Square Root”. In: *IEEE Transactions on Computers* 51.12 (2002), pp. 1377–1388 (cit. on pp. 13, 260, 288, 303).
- [PC16] Ayan Palchaudhuri and Rajat Subhra Chakraborty. *High Performance Integer Arithmetic Circuit Design on FPGA*. Vol. 51. Springer Series in Advanced Microelectronics. 2016 (cit. on p. 24).
- [Roj97] Raúl Rojas. “Konrad Zuse’s Legacy: The Architecture of the Z1 and Z3”. In: *IEEE Annals of the History of Computing* 19.2 (1997) (cit. on p. 1).
- [SWS00] Michael J. Schulte, Kent E. Wires, and James E. Stine. “Variable-Correction Truncated Floating Point Multipliers”. In: *Asilomar Conference on Signals, Circuits and Systems*. IEEE, 2000, pp. 1344–1348 (cit. on p. 10).
- [Tan89] Ping Tak Peter Tang. “Table-Driven Implementation of the Exponential Function in IEEE Floating-Point Arithmetic”. In: *ACM Transactions on Mathematical Software* 15.2 (1989), pp. 144–157 (cit. on p. 2).
- [Tan90] Ping Tak Peter Tang. “Table-Driven Implementation of the Logarithm Function in IEEE Floating-Point Arithmetic”. In: *ACM Transactions on Mathematical Software* 16.4 (1990), pp. 378–400 (cit. on p. 2).

- [Tan92] Ping Tak Peter Tang. "Table-Driven Implementation of the Expm1 Function in IEEE Floating-Point Arithmetic". In: *ACM Transactions on Mathematical Software* 18.2 (1992), pp. 211–222 (cit. on p. 2).
- [Tay12] Michael B. Taylor. "Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse". In: *Design Automation Conference (DAC)*. ACM/IEEE, 2012 (cit. on pp. 3, 4).
- [TK00] Naofumi Takagi and Seiji Kuwahara. "A VLSI Algorithm for Computing the Euclidean Norm of a 3D Vector". In: *IEEE Transactions on Computers* 49.10 (2000), pp. 1074–1082 (cit. on p. 11).
- [WBL06] Xiaojun Wang, Sherman Braganza, and Miriam Leeser. "Advanced Components in the Variable Precision Floating-Point Library." In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2006, pp. 249–258 (cit. on pp. 13, 260, 288, 301, 303).
- [Zus84] Konrad Zuse. *Der Computer – Mein Lebenswerk*. Springer, 1984 (cit. on p. 1).



2

CHAPTER 2

Number Formats

*There are 10 types of people in this world:
those who understand binary, and those who don't.*

Anonymous

A variety of number systems can be used to represent integer or real data in applications, and application-specific arithmetic begins with a choice of the most relevant ones. This chapter introduces these number systems and studies their essential properties.

This chapter introduces and centralizes information about the various number formats used in this book. It first presents the basics: the binary representations of integers in Sect. 2.1 and the fixed-point representation of real numbers in Sect. 2.2. These representations are generalized to high-radix in Sect. 2.3 and to redundant systems in Sect. 2.4. Finally, standard and non-standard floating-point formats are presented in Sect. 2.5 and alternative logarithmic representations in Sect. 2.6.

Some of these formats will already be familiar to some of our readers. In this case, browsing through the section titles and looking at the figures should be enough to get familiar with the notations used throughout this book, and the reader will know where to come back for reference when needed.

2.1 Representing Integers in Binary

A bit is an atom of information that can take only two values. For logic purposes these two values are called true and false; for arithmetic purposes, they are called 0 and 1. When mixing arithmetic and logic (for instance, to build logic circuits that compute the purpose of this book), the standard convention is to identify 0 to false and 1 to true.

2.1.1 Binary Representation of Positive Integers

A vector (x_{w-1}, \dots, x_0) of w bits ($w \in \mathbb{N}$) can take 2^w distinct values. It can therefore be used to encode distinctly all the integer in the interval $[0, 2^{w-1} - 1]$. The prevalent binary coding system¹ relates the values of the bits x_i and an integer $X \in [0, 2^{w-1} - 1]$ as follows:

$$X = \sum_{i=0}^{w-1} 2^i x_i. \quad (2.1)$$

Figure 2.1 is an illustration of this equation. It represents the value 25 as a vector of 8 bits, 3 of which are equal to ones as $25 = 2^4 + 2^3 + 2^0$.

bit weight	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
bit position	7	6	5	4	3	2	1	0
	0	0	0	1	1	0	0	1

Fig. 2.1 Representation of the positive integer 25 as an 8-bit vector.

Throughout this book, we often manipulate *bit positions*² such as the index $i \in \mathbb{N}$ in (2.1) (Sect. 2.2 will generalize positions to $i \in \mathbb{Z}$). To each bit position is associated a *bit weight* (or just *weight*), a power of two such as the 2^i in (2.1). We may say, for instance, that this equation represents X as a sum of weighted bits.

This representation of positive integers is often called (in particular in programming languages) an *unsigned* representation, by contrast with the signed representation that we introduce now. Unsigned really means positive or zero.

¹ This binary representation is the simplification to base 2 of the decimal positional number system that has been prevalent in human writing for several centuries.

² Many number systems are based on sums of weighted tokens; consider, for instance, a money amount expressed in coins and bills or the Roman numerals. When the weight of a token is based on its position, the system is called a positional number system. Positional number systems scale well with very few tokens and are easy to compute with.

2.1.2 Signed Integers in Two's Complement Representation

The binary representation of an integer X on w bits given by (2.1) shows its limits as soon as we attempt to perform additions with it. More precisely, if the sum of two w -bit numbers is strictly larger than 2^w , it cannot be represented on w bits. For instance, the sum of the two 4-bit numbers 15 and 2, respectively, represented in binary by 1111_2 and 0010_2 , is 17 which written in binary requires 5 bits : $17 = 10001_2$.

It is easy to show that the sum of two binary w -bit numbers can always be represented on $w + 1$ bits. If we have only w bits to store the result, we must discard this extra bit of weight w . The operation thus performed is then no longer the addition of integers. It is another simple mathematical operation, the addition modulo 2^w of integers in $[0, 2^w - 1]$. On our previous example, this new operation gives $1111_2 + 0010_2 = 0001_2$. The result is 1 which is indeed equal to 17 modulo 16.

The same is true for multiplication: the product of two w -bit unsigned integers requires up to $2w$ bits. If we discard the w most significand bits because we cannot store them, the operation implemented is actually the multiplication modulo 2^w of unsigned integers.

Thus, from an arithmetic point of view, a *binary representation on w bits is best understood as a representation of integers modulo 2^w* . For this reason, Fig. 2.2 represents all the 4-bit codes as a ring that illustrates the modulo 2^4 arithmetic.

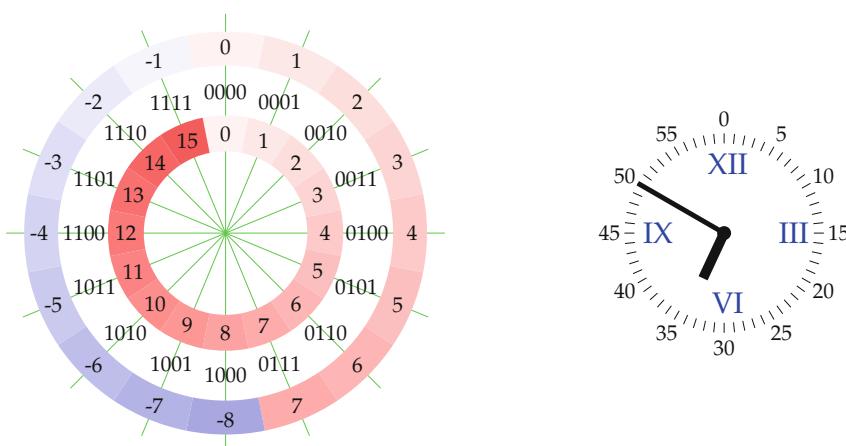


Fig. 2.2 Left: Binary 4-bit integers as a modulo-16 representation, with values given as unsigned (inner circle) or two's complement (outer circle). Right: parallel with the dial of an old-fashioned analog clock. When one says “10 minutes to 7” instead of “6 hour 50,” we similarly have two interpretations of the same position of the minute hand: -10 and 50 are equal modulo 60.

Then subtraction should also be considered as subtraction modulo 2^w . This brings in a sensible definition of the opposite of a binary number X on w bits: $-X$ is simply $0 - X$ modulo 2^w . It is more comfortable to rewrite it (modulo 2^w) as $2^w - X$: then the subtraction always yields a positive result for our interval $[0, 2^w - 1]$. For instance, on 4 bits, -1 can be represented modulo 2^4 as $2^4 - 1 = 1111_2$. The representation of -2 is $2^4 - 2 = 1110_2$, etc.

This is the essence of *two's complement representation* of negative integers (the name is historical), illustrated by Fig. 2.2.

It remains to decide, among the 2^w binary codes on w bits, which should code for positive numbers and which should code for negative ones. For instance, on 4 bits, 1110_2 can be interpreted as -2 or as 14, since these two values are identical modulo $2^4 = 16$. The convention for two's complement is that the codes whose most significant bit (MSB) is 1 will represent negative numbers and those whose MSB is 0 will represent positive numbers. Thus, we have about as many positive numbers as negative numbers, and the MSB serves as a sign bit s , as illustrated by Fig. 2.3.

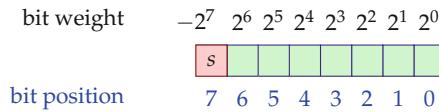


Fig. 2.3 Signed integer representation on 8 bits (two's complement or sign-magnitude).

Figure 2.2 gives all the values for 4-bit two's complement. In general, the interval of integers represented in two's complement on w bits is $[-2^{w-1}, 2^{w-1} - 1]$.

Beware that the interval $[-2^{w-1}, 2^{w-1} - 1]$ is not symmetrical around zero, due to the fact that zero, still coded by $00\dots00$, has a positive code. In other words, there is one negative value, -2^{w-1} , whose opposite is not representable. If not accounted for, this can lead to bugs that will be triggered by this specific value.

The beauty of two's complement is that if we have some operator for binary addition modulo 2^w (i.e., discarding the $w+1$ -th bit), then this operator can either serve as an adder for unsigned binary integer or for two's complement signed integers. Furthermore, in the latter case, this operator will work as well for any combination of positive or negative integers, as long as the result belongs to the interval $[-2^{w-1}, 2^{w-1} - 1]$ of representable integers. Otherwise we have an overflow situation. The management of overflow situations is the only difference between additions in unsigned binary and two's complement.

For illustration, let us compute $(-2) + (-1)$ in two's complement on 4 bits. The operator will compute $1110_2 + 1111_2$, which is (interpreted as unsigned) $14 + 15 = 29 = 01101_2$. Keeping only the 4 least significant bit (LSB) we get 1101_2 , which is the code of -3 .

Let us now give a few other essential properties of two's complement.

From the observation that the binary representation of $2^w - 1$ is a string of w ones,³ it is possible to relate the bitwise NOT of X (noted \bar{X}) to subtraction by the following identity:

$$\bar{X} = (2^w - 1) - X. \quad (2.2)$$

Indeed, the subtraction happens bitwise, and each bit subtraction computes $1 - x_i$ which is \bar{x}_i .

Using this relation, the opposite of a number X in binary two's complement on w bits can be computed by

$$-X = \bar{X} + 1 \mod 2^w. \quad (2.3)$$

For instance, on 4 bits, the negation of $1 = 0001_2$ can be computed as $\overline{0001}_2 + 1 = 1110_2 + 1 = 1111$ which is indeed the code of -1 . It works both ways: the opposite of -1 can be computed as $\overline{1111}_2 + 1 = 0000_2 + 1 = 0001$. Our reader will check that the opposite of 0 is still 0 (and this case triggers the modulo 2^w).

Finally, the value of an integer X in two's complement on w bits is

$$X = -2^{w-1}x_{w-1} + \sum_{i=0}^{w-2} 2^i x_i. \quad (2.4)$$

In other words, each bit has the same weight as in the unsigned case, except for the MSB where this weight is negative.

2.1.3 Sign-Magnitude Representation

It may seem simpler to represent an integer as two fields: a sign bit s and an unsigned binary number on $w - 1$ bits. When defining a single bit vector for these two fields, the sign bit will be placed at the MSB as shown in Fig. 2.3. Such a sign-magnitude representation matches the usual decimal convention of writing the sign to the left of the magnitude as in “ -17 .”

With the same convention as in two's complement that $s = 0$ for positive and $s = 1$ for negative numbers, the value of a sign-magnitude fixed-point number corresponding to Fig. 2.3 is thus

³ This is simply the transposition to binary of the decimal identity $10^4 - 1 = 9999$ which should be familiar to all our readers.

$$X = (-1)^s \times \sum_{i=0}^{w-2} 2^i x_i \quad (2.5)$$

where $(-1)^s$ has the value +1 when $s = 0$ and -1 when $s = 1$.

Hence, the range of representable numbers is $X \in [-2^{w-1} - 1, 2^{w-1} - 1]$. It has the nice property that it is symmetrical around zero. However, zero has two representations (+0 and -0), which complicates equality test.

Sign-magnitude is mostly used in the representation of floating-point numbers (discussed in Sect. 2.5 below). It is almost never used for integers, where two's complement is much more elegant, leading to simpler hardware.

2.1.4 Conversion Between Sign-Magnitude and Two's Complement

The sign convention, together with the use of x_{w-1} as the sign bit s , leads to the following property: the value of the sign-magnitude representation of a positive integer is the same as if these bits are interpreted as an unsigned integer.

Conversely, negative numbers are encoded differently in sign-magnitude and in two's complement. In this case (when the sign bit is set), the conversion of one format to the other is, either way, a two's complement negation:

- To convert a negative sign-magnitude number to two's complement number, the $w - 1$ -bit magnitude is first rewritten as a w -bit positive two's complement number of the same value (by adding a 0 MSB). Then the opposite of this value in two's complement is computed as $\bar{X} + 1$.
- To convert a negative two's complement number to sign-magnitude, the opposite of this number is computed as $\bar{X} + 1$. This process resets the sign bit to zero, so the $w - 1$ LSBs are the magnitude of the result. The sign of the result must then be restored.

Note that the second conversion will fail on one value: -2^{w-1} .

2.2 Fixed-Point Binary Representations

2.2.1 Unsigned Fixed-Point Binary Representation

Equation (2.1) can be straightforwardly generalized to fractional numbers. Two integers $m \in \mathbb{Z}$ and $\ell \in \mathbb{Z}$ such that $m \geq \ell$ define a *fixed-point unsigned format* denoted as $\text{ufix}(m, \ell)$ and illustrated by Fig. 2.4. This format is the set of rational numbers X that can be written as

$$X = \sum_{i=\ell}^m 2^i x_i \quad . \quad (2.6)$$

Here, a bit position may be positive or negative. However, the value of X is always positive or zero. The “u” in ufix stands for “unsigned,” which again really means “positive or zero.”

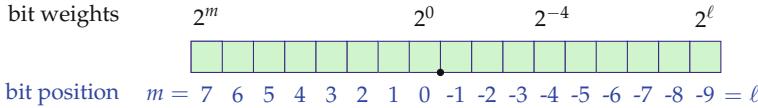


Fig. 2.4 The unsigned fix-point format ufix(m, ℓ).

The bits with positive positions, if any, belong to the integer part of X . The bits with negative positions, if any, belong to the fractional part of X : their weight is smaller than one ($2^{-1} = 1/2 = 0.5, 2^{-2} = 1/4 = 0.25$, and so on). For the unfamiliar reader, this is nothing more than the transposition to binary (base 2) of the familiar decimal (base 10) notation: for instance, in the decimal number 3.14, the decimal weight of the digit 1 is $10^{-1} = 0.1$, the decimal weight of the digit 4 is $10^{-2} = 0.01$, and the value of 3.14 is $3 \cdot 10^0 + 1 \cdot 10^{-1} + 4 \cdot 10^{-2}$. Similarly, in binary, the value of 11.001001 is $1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-3} + 1 \cdot 2^{-6}$.

The integer ℓ denotes the position of the LSB of X and defines its *precision*, or its *resolution*. Indeed, the minimal distance between two successive numbers in ufix(m, ℓ) is 2^ℓ . This value 2^ℓ is the weight of the LSB. It is sometimes called the quantization step of the format, or its unit in the last place (ulp).

The integer m denotes the position of the MSB of X . It defines the *range* of the format ufix(m, ℓ). Indeed, the maximum value that X can take is $2^{m+1} - 2^\ell$ and usually $2^\ell \ll 2^{m+1}$.

An ufix(m, ℓ) number requires $w = m - \ell + 1$ bits for its representation.⁴ Its dynamic range, which is defined as the ratio between the largest number $2^{m+1} - 2^\ell$ and the smallest non-zero number 2^ℓ , is $2^{m-\ell+1} - 1 = 2^w - 1$.

⁴ With the condition $m \geq \ell$, a fixed-point number has at least one bit. In a software context, it may help simplify the implementation to allow data formats whose bit-width is zero. A number in such a format always has the value zero.

What does “precision” mean?

The reader should be aware that a phrase like “the precision of a format” is quite ambiguous. It may mean 2^ℓ , which is the quantum of representation. “The precision of the measure has been doubled” probably means that one bit of information has been gained. However it may also mean that the number of bits has been doubled. Indeed, “double-precision” describes a floating-point format with twice as many bits as “single precision.” These are not the numbers of significand bits, though (respectively, 52 and 23). The exponent bits should not be counted in the precision, but they are in this case. As “precision” means “number of bits” in this floating-point case, this meaning has contaminated fixed-point as well: sometimes, the precision of a fixed-point format will mean its total size in bits, although “The precision of the measure has been doubled” probably means more bits on the right, not on the left of the number.

The defense against such ambiguities is first to be aware of them and then to attempt to complement ambiguous words with clarifying adjectives or mathematical notations, e.g., “the precision ℓ_X of the signal X ” is better than “the precision of the signal X .”

There are two particularly important cases of unsigned fixed-point formats:

- The format $\text{ufix}(w - 1, 0)$ denotes unsigned binary integers on w bits, as seen in the previous section (see Fig. 2.5).
- The format $\text{ufix}(-1, -w)$ defines numbers in $[0, 1)$ on w bits.

Also note that (2.6) can be rewritten

$$X = \sum_{i=\ell}^m 2^i x_i = 2^\ell \sum_{i=0}^{m-\ell} 2^i x_{i+\ell} . \quad (2.7)$$

This shows that an $\text{ufix}(m, \ell)$ fixed-point number can also be viewed as an unsigned integer of $m - \ell + 1$ bits, scaled by a constant power of two 2^ℓ . The weight 2^ℓ of the LSB is therefore also sometimes called scaling factor.

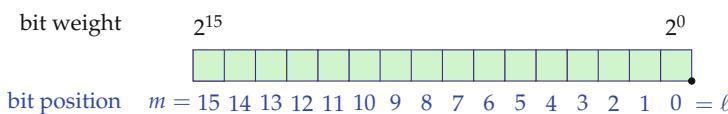


Fig. 2.5 An unsigned 16-bit integer is $\text{ufix}(15, 0)$.

2.2.2 Two's Complement Fixed-Point Binary Representation

Two's complement integers can similarly be scaled. A signed fixed-point format of MSB m and LSB ℓ (see Fig. 2.6) in two's complement is denoted by $\text{sfix}(m, \ell)$. The value X of a number in this format is defined by the following equation, which is the scaled version of (2.4):

$$X = -2^m x_m + \sum_{i=\ell}^{m-1} 2^i x_i . \quad (2.8)$$

The sign bit now has weight 2^m and the range of representable numbers in $\text{sfix}(m, \ell)$ is $X \in [-2^m, 2^m - 2^\ell]$. Again this interval is not symmetrical around zero.

Note that the condition $m \geq \ell$ holds as in the unsigned case. In the case $m = \ell$, only the sign bit remains: a 1-bit signed number can take only two values, 0 and -2^m .

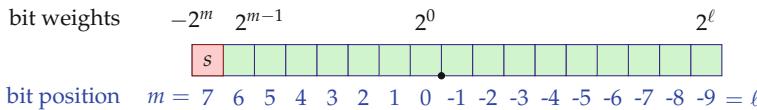


Fig. 2.6 Signed fixed-point formats ($\text{sfix}(m, \ell)$) or sign-magnitude).

Two families of signed fixed-point formats are of particular interest:

- The format $\text{sfix}(w-1, 0)$ denotes two's complement binary integers on w bits, having a range $X \in [-2^{w-1}, 2^{w-1} - 1]$.
- The format $\text{sfix}(0, -w+1)$ defines numbers in $[-1, 1]$ on w bits: the sign bit has weight 2^0 , and the LSB has weight 2^{-w+1} . An example is depicted in Fig. 2.7.

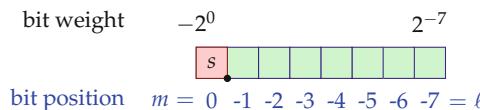


Fig. 2.7 $\text{sfix}(0, -7)$, an 8-bit fixed-point format for numbers in $[-1, 1 - 2^{-7}]$.

As in the unsigned case, an $\text{sfix}(m, \ell)$ fixed-point number can be viewed as a signed integer of $w = m - \ell + 1$ bits, scaled by 2^ℓ .

A variant of the fixed-point signed formats called Half-Unit Biased or HUB will be presented in Sect. 3.1.6 of the next chapter.

2.2.3 Conversion to a Wider Format: Sign Extension

Just like in decimal 17.42 is equal to 0017.42000, an unsigned fixed-point number in some format $\text{ufix}(m, \ell)$ can be converted to a wider format $\text{ufix}(m', \ell')$ (where wider means that $m' > m$ or $\ell' < \ell$) by padding left and right with zeroes. This conversion is errorless and in most situations for free.

Conversion of a *signed* format $\text{sfix}(m, \ell)$ to a wider format $\text{sfix}(m', \ell')$ is slightly less straightforward. It also involves padding right with $\ell - \ell'$ zeroes, since the weighted value of any of these zeroes in (2.8) is 0. When $m' > m$, however, padding left with $m' - m$ zeroes changes the values of negative numbers (they even become positive, since the leftmost of the padded zeroes becomes the sign bit of the wider format). Errorless conversion requires instead to pad negative numbers with ones. Positive numbers still need to be left-padded with zeroes. So in general, a negative number can be padded left by the sign bit without changing its value. This transfers the sign bit from position m to position m' thanks to the identity $-2^m = -2^{m'} + \sum_{i=m}^{m'-1} 2^i$.

In summary, the errorless conversion from $\text{sfix}(m, \ell)$ to a wider format $\text{sfix}(m', \ell')$ consists in right-padding with zeros while left-padding with copies of the sign bit (a process usually named *sign extension*). See Fig. 2.8 for an illustration.

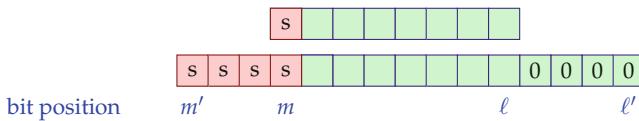


Fig. 2.8 Sign extension of a two's complement number from format $\text{sfix}(m, \ell)$ to a wider format $\text{sfix}(m', \ell')$. The two represented numbers have the same value.

2.2.4 Conversion to a Narrower Format: Overflow and Rounding

Narrowing the format, i. e., from $\text{sfix}(m, \ell)$ to $\text{sfix}(m', \ell')$ with $m' < m$ or $\ell' > \ell$, is often unavoidable, as an exact computation tends to return a results that requires more bits than its inputs. For instance, the exact product of two w -bit numbers is a $2w$ -bit number. Worse, the natural logarithm of the 2-bit integer 2 is an irrational number which would require an infinite number of bits to be represented exactly in binary.

Contrary to conversions to wider formats, narrowing conversions thus involve some information loss. One must distinguish what happens at the MSB and at the LSB.

Narrowing at the MSB ($m' < m$) means that some numbers in $\text{sfix}(m, \ell)$ are out of range for $\text{sfix}(m', \ell')$. They are not representable at all, and even the order of magnitude is lost. This is called an overflow situation and is potentially catastrophic: it should be avoided.

In this book, chapters describing fixed-point operators usually begin with a worst-case analysis defining the minimum value of their output MSB position m that ensures that the result value will not overflow, whatever the inputs. However, this generic analysis only consider the operator in isolation. It is often the case that the context provides more information that permits the use of a smaller (context-specific) m . It may be some mathematical knowledge of the problem at hand, or some correlation on the inputs of the operator, or some application-specific constraint on the input ranges, etc. Several examples of such safe MSB narrowing will be given in this book.

Narrowing at the LSB ($\ell' > \ell$) means that some accuracy will be lost when converting from $\text{sfix}(m, \ell)$ to $\text{sfix}(m', \ell')$. This can be achieved by truncation (simply dropping bits) or by more accurate rounding techniques such as round to the nearest that are detailed in the next chapter. Contrary to overflow, rounding (sometimes also called quantization) is often inevitable. It is also very often acceptable: indeed one purpose of this book is to provide tools to keep the accuracy of the overall computation under control in the presence of rounding. This will be achieved by controlling the LSB parameter ℓ of the format of each intermediate computation. As a larger ℓ usually entails cheaper and faster operators (fewer bits to compute), this parameter will be used as a cursor to control the accuracy versus performance trade-off.

2.2.5 Alternative Notations for Fixed-Point Formats

The notations chosen in this book were inspired by the standard IEEE `fixed_pkg` package used to specify fixed-point numbers in the VHDL language. There are several different but equivalent notations for fixed-point numbers. Among them,

- The “Qi.f” notation⁵ defines a fixed-point format by two integers, i denoting the “number of integer bits” (equal to $m + 1$), and f the “number of fractional bits” (equal to $-\ell$).
- The two main C++ libraries used for high-level synthesis (HLS), `ap_fixed` and `ac_fixed`, share a third approach: they again use two integers to define a fixed-point type, the total size w (equal to $m - \ell + 1$) and the “number of integer bits” (equal to $m + 1$).

⁵ It is unclear if the Q stands for “quantized” or for the mathematical symbol Q denoting the set of rationals (but then remark that any integer, fixed-point or floating-point number manipulated in a machine is also a rational number).

These notations are equivalent to the one used in this book, assuming the “number of bits” above are allowed to become negative, which we consider counterintuitive (hence the quotes used). For instance, consider a signal strictly bounded by 2^{-5} . It can use, for example, the format $\text{ufix}(-6, -16)$, which reads as follows: bits range from positions -16 to position -6 (both included). This is not a toy example, such cases actually appear in Chaps. 9, 12, 17, 18, 20, 22, and 23 of this book. The format $\text{ufix}(-6, -16)$ can also be expressed as Q-5.16 , but then both “number of bits” of the Qi,f notation become awkward: “minus 5 integer bits” makes little sense, and the actual “number of fraction bits” is not 16 as written, but 11. This is one good reason to prefer the proposed notation, where both parameters are bit positions which can be either positive or negative. It turns out that the chosen notation also makes for simpler formulations for the purpose of this book. For instance, (2.6) becomes more complex with any of the other notations. Finally, when attempting to finely tune arithmetic operators, it helps to attach a binary weight to each bit, and the chosen notation makes this straightforward.

This being said, viewing fixed-point numbers as integers scaled by 2^ℓ is very productive when building fixed-point operators, as it allows to reduce them to integer ones and constant shifts (i. e., wires). It is also the proper way to describe fixed-point computations on a processor that only offers integer units.

Thus, the preferred notation is often a matter of taste, habit, or context. The main point here is that conversion between any of these notations is straightforward.

Some of the libraries mentioned above also specify, for a fixed-point type, its rounding and overflow behavior in case a narrowing conversion is needed. These will be discussed in Sect. 3.1.5 of the next chapter.

2.3 High-Radix Binary Representation

Our reader is probably already familiar with the *hexadecimal* notation for binary data. As illustrated by Fig. 2.9, it consists in grouping the bits of the data in 4-bit chunks. Then it is possible to map to each 4-bit chunk an integer between 0 and 15, with the values 10 to 15 denoted with the letters A to F.

F	2	D
1 1 1 1	0 0 1 0	1 1 0 1

Fig. 2.9 Hexadecimal is an example of high-radix binary representation.

Hexadecimal is pervasively used in computing as a compact representation of binary data (which is not necessarily numerical: it is, for example,

also a representation of choice for the machine instructions when working with executable object files). Hexadecimal can also be used, as the name suggests, as a radix-16 representation of binary integers. Here, the radix is $16 = 2^4$ (instead of 10 in decimal), the digits to be used are the 16 digits from 0 to $F = 15$ (instead of the 10 digits from 0 to 9), and an hexadecimal number of n digits $X_{n-1} \dots X_0$ represents the value

$$X = \sum_{i=0}^{n-1} 16^i X_i . \quad (2.9)$$

Since (2.9) differs from the usual decimal representation only in the value of the radix, it is possible to perform operations on this representation by adapting decimal algorithms. For instance, Fig. 2.10 shows an addition performed in hexadecimal. It starts with $D_{16} + A_{16} = 13_{10} + 10_{10} = 23_{10} = 16_{10} + 7_{10} = 17_{16}$: the sum of these two rightmost digits is written 17 in hexadecimal, where the 1 is a carry that is added to the column at the left. The next step is to compute the sum $1 + D + 5$ and write it in hexadecimal and so on. We invite our reader to complete this addition as an exercise.

$$\begin{array}{r} \text{FDD} \\ + \quad \text{A5A} \\ \hline \text{111} \\ \hline \text{1A37} \end{array}$$

Fig. 2.10 Hexadecimal addition.

What this example illustrates is that hexadecimal is not only a data format, but a full-blown number representation system with its range of arithmetic operations. Hexadecimal is but a particular case of what we call *high-radix binary representations*.⁶ Given an integer parameter $\alpha > 1$, grouping in chunks of α bits the bits of an integer X of size $w_X = k\alpha$ bits (for some integer k) defines the radix- 2^α representation of a binary number:

$$\begin{aligned} X &= \sum_{i=0}^{w_X-1} x_i \cdot 2^i \\ &= \sum_{i=0}^{k-1} X_i \cdot (2^\alpha)^i \text{ where } X_i = \sum_{j=0}^{\alpha-1} 2^j x_{j+\alpha i} . \end{aligned} \quad (2.10)$$

⁶ Remark that high-radix *decimal* is also something we are used to, for instance, when expressing 1985 as “nineteen hundred and eighty five”: grouping decimal digits in chunks of 2 yields a radix-100 representation.

Here, X_i is the binary value of the subword $X[\alpha i, \dots, \alpha i + \alpha - 1]$ and therefore $X_i \in \{0, \dots, 2^\alpha - 1\}$. The equality between the two lines of (2.10) is straightforward.

Just like hexadecimal, this high-radix binary representation is really just a change of point of view on a binary number. In terms of hardware, grouping bits in chunks is for free.

Nevertheless, it is a very useful tool to generalize arithmetic algorithms, as the parameter α defines the granularity of the number representation. For instance, a radix-32 representation ($\alpha = 5$) has fewer digits than a radix-8 one ($\alpha = 3$), but each digit holds more information; therefore the elementary digit operations will be more complex. In Chap. 9, for instance, this will be exploited to expose a trade-off between the number of iterations of a division algorithm and the hardware complexity of each iteration.

Conversely, the context may impose a granularity. In particular, when the hardware target is an FPGA, the logic granularity is defined by the size of the architectural look-up table (LUT) (see Sect. 4.1). The large LUTs of modern FPGAs enable efficient operations on relatively large digits. This will be exploited to implement efficient multiplications and divisions by constant in Chaps. 12 and 13, respectively. Another example, still on FPGAs, will be given in Chap. 21 where the granularity will be defined as the maximum size of an addition that can be performed in one cycle.

As a final remark, we have defined here the high-radix binary representations on integers, but it can also be used, *mutatis mutandis*, on any fixed-point signed or unsigned format.

2.4 Redundant Positional Number Systems

The fixed-point binary, the usual decimal system, and the high-radix formats of previous sections are all instances of positional number systems. Such a system is defined by a parameter, the base or radix, denoted as β in this book. We have $\beta = 2$ for binary, $\beta = 10$ for decimal, and $\beta = 2^\alpha$ for high-radix binary. A fixed-point format in a radix- β position system is defined by two more integers m and ℓ such that a number X is represented by a sequence of $m - \ell + 1$ digits $x_i \in \{0, \dots, \beta - 1\}$, with the identity

$$X = \sum_{i=\ell}^m \beta^i x_i . \quad (2.11)$$

It is possible to define more number systems by changing the *digit set*, i. e., the set of possible values for the x_i . Indeed, any set of at least β consecutive integers that includes zero can be used as a digit set for radix- β . Let us first take a decimal example from history.

Cauchy showed in 1840 [Cau40] that a decimal system could use the digit set $\mathcal{D} = \{\bar{5}, \bar{4}, \bar{3}, \bar{2}, \bar{1}, 0, 1, 2, 3, 4, 5\}$, where the digits $\bar{5}$ to $\bar{1}$ have the respective values -5 to -1 . In this system, the value of a number X is still defined as $X = \sum 10^i x_i$, with the digits $x_i \in \mathcal{D}$: it is therefore a decimal system. For instance, the value 17 is written $\bar{2}\bar{3}$ in this system ($\bar{2}\bar{3} = 20 - 3$). One advantage of such a system is that negative numbers are included, without the need of a minus sign, e. g., -17 is written $\bar{2}\bar{3}$. Since our addition, multiplication and division algorithms are derived from the formula $X = \sum 10^i x_i$, they still work with only marginal adaptation (you will have both positive and negative carries in addition, for instance).

The initial motivation of Cauchy was to have a completely different system in which complex computations could be performed a second time, so that their result can be independently checked for errors. However, he found several other advantages to his signed-digit system:

- The multiplication table is reduced to multiplications up to 5×5 (and a trivial sign rule). This is one quarter of the usual table and the easiest to memorize. We will see in Chap. 9 a transposition of this observation to high-radix binary which can similarly help simplify the computation of intermediate products in a division algorithm.
- Cauchy also observed that carry generation was much rarer, in particular in large summations (e. g., in the schoolbook multiplication algorithm), where positive and negative carries would cancel. Later, Avižienis [Avi61] independently showed that carry propagation could be avoided altogether.

An important remark is that Cauchy's system is *redundant*. For instance, 15 can be written 15 ($10 + 5$), or $\bar{2}\bar{5}$ ($20 - 5$). This is actually the key to avoiding carry propagation: when the sum of two digits could possibly propagate an incoming carry, choose for this digit sum the representation that will absorb any incoming carry. For instance, in Cauchy's system, $4321 + 1235$ is a situation that entails a carry propagation: the sum of the two digits in the rightmost columns is $5 + 1 = 6$ which we have to rewrite as $\bar{1}\bar{4}$, thus creating a carry. This carry will propagate to the other columns, as the sum of each of these columns is 5. However, if we chose instead to first rewrite each digit as $5 = \bar{1}\bar{5}$, then an incoming positive carry will transform the $\bar{5}$ into a $\bar{4}$. The full addition is thus performed in two steps as follows: $4321 + 1235 = 11111 + \bar{5}\bar{5}\bar{5}\bar{4}$ (anticipated generation of all possible carries) then $11111 + \bar{5}\bar{5}\bar{5}\bar{4} = \bar{4}\bar{4}\bar{4}\bar{3}$ (carry propagation to at most one digit). Both steps can be performed in a digit-parallel fashion (or left to right, in MSB-first digit-serial arithmetic). This was just an example, but it can be generalized into an algorithm without carry propagation [Avi61]. Variants of this algorithm will be used in several chapters of this book (and defined properly then).

We could use an even larger digit set, for instance, $\{\bar{7}, \bar{6}, \dots, \bar{1}, 0, 1, ..7\}$. In this new set, we have even more redundancy, for instance, all the integers between 13 and 17 can now be written in two different ways (e.g., $13 = \bar{2}\bar{7}$).

Redundancy not only helps avoiding carry propagations (which is exploited in Chaps. 5 and 7); it also provides representation freedom that will help optimize multiplication (Chap. 8), division (Chap. 9), and square root and other algebraic functions (Chap. 19).

We do not cover it in this book, but there has also been a lot of research [Erc77; Mul94; GT04; Li+18; Ska+20] on *online arithmetic* [EL04], where the digits of a number are transmitted serially, most significant digit first, from operator to operator. Online arithmetic may use radix 2 with the redundant digit set $\{\bar{1}, 0, 1\}$, or a larger radix. It is then possible to build an operator that begins outputting digits of its result as soon as it has input a few digits of its arguments. One potential advantage of online arithmetic is thus the inherent ability of operators to work with numbers of various sizes.

However, redundancy also entails a cost. Firstly, more bits are usually needed to represent each digit compared to a non-redundant digit set in the same radix. Secondly, conversion to a non-redundant digit set involves either a carry propagation, or a MSB-first digit recurrence [EL04]. In this sense, redundant formats can be viewed as carry-delays techniques.

2.5 Binary Floating-Point Formats

The dynamic range of fixed-point numbers is a limiting factor that may require excessively large word sizes for some applications. *Floating-point* numbers provide a much wider dynamic range, at the cost of a reduced precision, for the same number of bits.

This section is a brief introduction to two families of binary floating-point formats. It starts with the IEEE 754 standard format, supported by most computers. A simplified version, widely used in application-specific arithmetic, will then be presented in Sect. 2.5.3, and the advantages and drawbacks of both approaches will be discussed in Sect. 2.5.4.

2.5.1 The IEEE 754 Formats

This family of formats was standardized in 1985 [754-85], with standard revisions in 2008 [754-08] and 2019 [754-19]. The 2008 revision integrated decimal arithmetic, but we choose not to discuss it in this book: all the following addresses only binary IEEE 754 floating-point.

A floating-point number X according to IEEE 754 is represented by three fields s , E , and F as shown in Fig. 2.11.

- s is a sign bit (0 for positive, 1 for negative).
- E is an exponent field of w_E bits, interpreted as a positive integer.
- F is a fraction field of w_F bits, interpreted as a fixed-point number in the $\text{ufix}(-1, -w_F)$ format: it is a fractional number in the interval $[0, 1]$.

Therefore, a floating-point format is essentially defined by two parameters: w_E the size of the exponent field and w_F the size of the fraction field.

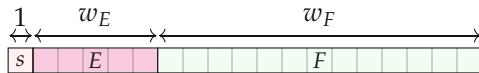


Fig. 2.11 Bit fields of an IEEE floating-point number $\text{IEEEfloat}(w_E, w_F)$.

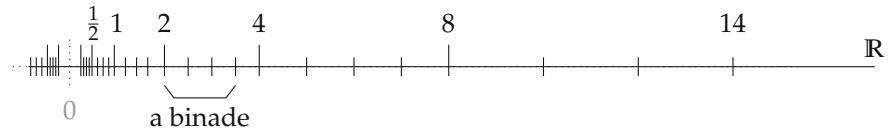


Fig. 2.12 Representation on the real axis of the normal numbers in a small IEEE 754-like format: $(w_E, w_F) = (3, 2)$. Longer bars are powers of two. As the set of normal number is symmetrical around zero, only the positive numbers are all shown. The maximum representable value in this case is $1.11_2 \cdot 2^3 = 14$. Note the gap around zero.

Such a format will be denoted as $\text{IEEEfloat}(w_E, w_F)$ in this book, even when we use values for w_E and w_F which do not correspond to any of the standard IEEE 754 formats.

In such a format, there are three categories of floating-point numbers, depending on the value of the exponent field E .

2.5.1.1 Normal Numbers

When E is neither 0 nor $2^{w_E} - 1$ (its two possible extremal values), the number represented is a *normal number*. Normal numbers are represented on the real axis in Fig. 2.12 for a small IEEE-like format. The value of a normal number is

$$X_{\text{normal}} = (-1)^s \times (1 + F) \times 2^{E - E_0} \quad (2.12)$$

where E_0 , the *exponent bias*, is a constant of the format whose value is

$$E_0 = 2^{w_E - 1} - 1 \quad . \quad (2.13)$$

Written in binary, E_0 has the form 011...11.

For normal numbers, the value of E belongs to the interval $[1, 2^{w_E} - 2]$; therefore, the actual exponent $E - E_0$ belongs to the interval $[e_{\min}, e_{\max}]$ where

$$e_{\min} = -2^{w_E-1} + 2 \quad (2.14)$$

and

$$e_{\max} = 2^{w_E-1} - 1 . \quad (2.15)$$

The value of the actual significand $1 + F$ lies in the interval $[1, 2)$. The 1 in $1 + F$ is called the *implicit bit*. Without it, some values could have several different representations (for instance, $1.00_2 \times 2^0 = 0.10_2 \times 2^1$) which would complicate comparisons. To avoid this, significands are *normalized* so that their leftmost digit is non-zero. In binary, if the leftmost bit is not a 0, then it is necessarily a 1; therefore it does not need to be stored.

2.5.1.2 Subnormal Numbers

The minimum value of the exponent field $E = 0$ is used to represent so-called *subnormal* numbers and zeroes. Subnormal numbers are represented in Fig. 2.13. They have the same exponent as the smallest normal numbers: $e_{\min} = -2^{w_E-1} + 2$, but the integer bit of the significand is 0: the value represented is

$$X_{\text{subnormal}} = (-1)^s \times (0 + F) \times 2^{e_{\min}} . \quad (2.16)$$

Zero is actually a particular case of subnormal, when $F = 0$.



Fig. 2.13 Zoom on Fig. 2.12, showing in red the subnormal numbers for $(w_E, w_F) = (3, 2)$. Note that the two signed zeroes come as a special case of subnormal.

Due to the sign bit, we actually have two zeros, $+0$ and -0 . The choice of having two signed zeroes is controversial: there is only one zero in mathematics. Its main advantage is that these two zeroes are the reciprocals of the two signed infinities. Otherwise, the IEEE 754 standard states that the two zeroes are compared equal. It is therefore safer in general to think of -0 as simply 0 than to think that it represents a very tiny negative value.

The following generic formulation captures normal numbers, subnormals, and zero by defining a “is normal” bit n :

$$n = \begin{cases} 0 & \text{if } E = 0 \\ 1 & \text{otherwise} \end{cases} \quad (2.17)$$

$$X = (-1)^s \times 2^{E-E_0+1-n} \times (n+F) \quad . \quad (2.18)$$

2.5.1.3 Exceptional Numbers: Infinity and Not A Number

The maximum value $2^{w_E} - 1$ of the exponent field E (binary 11...11) is used to encode the special values $\pm\infty$ and Not a Number (NaN). NaN is used for the result of operations such as $0/0$ or $\infty - \infty$. The special value is $\pm\infty$ when $F = 0$ (the sign bit determines the sign of infinity); otherwise it is NaN (for any $F \neq 0$).

NANs may be quiet or signaling, and they may carry a payload in their sign and significand bits [754-19; Mul+18]. These details are irrelevant in the present book.

2.5.1.4 Motivation for the Biased Exponent

The choice of encoding signed exponent using a bias, instead of using two's complement, brings a very nice property illustrated by Table 2.1: the order of positive floating-point numbers, from 0 to $+\infty$, is the order of their binary representation when interpreted as an integer. This property even holds for subnormals. This makes for simple implementations for operations such as comparison or computation of the next or previous floating-point number.

Table 2.1 Encoding of IEEE 754 binary16 (half-precision) positive numbers.

Sign	Exponent	Fraction	Value	Comment
0	00000	0000000000	0	Positive zero
0	00000	0000000001	$0.0000000001 \cdot 2^{e_{\min}}$	Smallest positive (subnormal)
	⋮		⋮	
0	00000	1111111111	$0.1111111111 \cdot 2^{e_{\min}}$	Largest subnormal
0	00001	0000000000	$1.0000000000 \cdot 2^{e_{\min}}$	Smallest normal
	⋮		⋮	
0	01111	0000000000	$1.0000000000 \cdot 2^0$	1
	⋮		⋮	
0	11110	1111111111	$1.1111111111 \cdot 2^{e_{\max}}$	Largest normal
0	11111	0000000000	$+\infty$	
0	11111	0000000001	NaN	Not a Number
	⋮		⋮	
0	11111	1111111111	NaN	Not a Number

2.5.1.5 Standard Formats

The IEEE 754-2008 standard actually explicitly defines four formats, with the parameters given by Table 2.2. All these formats are instances of the generic format defined above.

Table 2.2 Some properties of the formats defined by the IEEE 754-2019 standard.

IEEE name Old name C name	binary16 Half-precision	binary32 Single precision float	binary64 Double precision	binary128 Quad precision double
Total size	16 bits	32 bits	64 bits	128 bits
w_F	10	23	52	112
2^{-w_F}	$\approx 10^{-3}$	$\approx 3 \cdot 10^{-8}$	$\approx 2 \cdot 10^{-16}$	$\approx 2 \cdot 10^{-34}$
w_E	5	8	11	15
e_{\min}, e_{\max}	-14, +15	-126, +127	-1022, +1023	-16382, +16383
Smallest subnormal	$\approx 6.0 \cdot 10^{-8}$	$\approx 1.4 \cdot 10^{-45}$	$\approx 4.9 \cdot 10^{-324}$	$\approx 6.5 \cdot 10^{-4966}$
Smallest normal	$\approx 6.0 \cdot 10^{-5}$	$\approx 1.2 \cdot 10^{-38}$	$\approx 2.2 \cdot 10^{-308}$	$\approx 3.4 \cdot 10^{-4932}$
Largest	$\approx 6.6 \cdot 10^4$	$\approx 3.4 \cdot 10^{38}$	$\approx 1.8 \cdot 10^{308}$	$\approx 1.2 \cdot 10^{4932}$

2.5.2 Emerging Non-standard Formats for Machine Learning

Many new floating-point formats have been introduced recently to match the specific requirements of machine learning applications. They are only listed here; more details on machine learning will be provided in Chap. 24. Figure 2.14 compares them with the standard IEEE 754 formats.

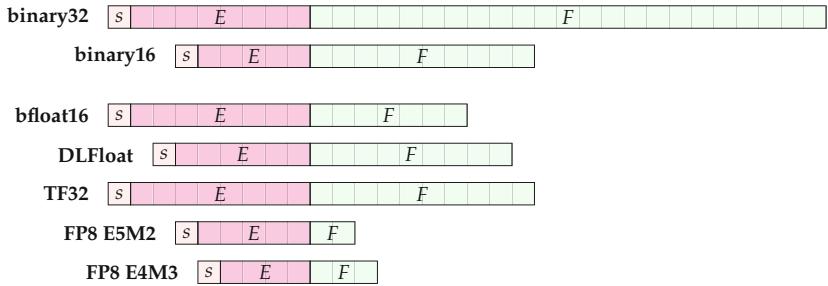


Fig. 2.14 Bit fields of non-standard formats used in machine learning.

bfloat16 (for *brain float*) is a 16-bit format adopted by Google, Intel [HTH19], ARM [Bur+19], and NVIDIA [NVI20]. This IEEEfloat(8,7) format is best understood as the upper half of a binary32 (single-precision) number: it consists of its sign bit, its 8 exponent bits, and the 7 leading bits of

its fraction. This gives it the same dynamic range as binary32, ensuring consistent overflow/underflow behavior when exchanging data with systems processing binary32.

DLFLOAT is an IEEEfloat(6,9) format proposed by IBM as the best trade-off between accuracy and dynamic range for machine learning applications among 16-bit IEEE-like formats [Agr+19].

TF32 (for TensorFloat-32) is not a 32-bit format. It is an IEEEfloat(8,10) format supported in hardware by NVIDIA A100 [NVI20]. It provides the dynamic range of binary32 like bfloat16, but also the precision of binary16. Thus, hardware support of this format benefits both binary16 and binary32 applications.

A wide consensus has emerged on two **FP8** formats [Sun+19; Dar+20; Mic+22; Nou+22] that together seem to fill the need of most machine learning tasks, even for very large models [Mic+22]. The need for two different formats will be explained in Chap. 24. E5M2 is an IEEEfloat(5,2): it is to binary16 what bfloat16 is to binary32. E4M3 is an IEEEfloat(4,3) modified as follows: it only has two NaN values (coded by s111111 where s is the sign bit), and it does not code $\pm\infty$, as infinities are considered useless in the specific context of machine learning inference. Renouncing NaN payloads and infinities makes space for almost one binade (7 values for each sign). This E4M3 format still includes subnormals (also 7 values for each sign). These two FP8 encodings are supported as input/output formats in the NVIDIA H100 TensorCore architecture [NVI22].

Note that we describe all these formats as IEEEfloat(w_E, w_F) because their encodings generalize the IEEE formats, but they do not belong to the IEEE 754 standard at the time of writing this book.

Many other variations on IEEE formats have been suggested. Some works consider adapting the bias value to the needs of the computation [Tam+19; Sun+19] or a shared exponent for a block of data (block floating point) [Dru+18; Dar+20; ZMK22].

Further away from IEEE 754, the *posit* format [GY17; Gro18; Din+19] is an encoding of floating-point numbers which trades significand bits for exponent bits when the exponent is small. Its use has been studied for machine learning tasks among others [LLH18; Car+19]. Hardware posit operators are more expensive than their IEEE 754 counterparts due to the overhead of posit encoding and decoding [FUD21].

It is important to understand that 8-bit formats are input/output formats only: the computations internally use larger precisions. Actually, when the exponent range is very small (16-bit and 8-bit formats), it becomes very cheap to compute the *exact* sum of n products (without any intermediate rounding error) for values of n which can be quite large. This technique is described in detail in Chap. 21 and is commonly used in recent machine

learning accelerators. In other words, although these formats are very imprecise, computations on them can become very accurate and even exact. This paradigm is one of the reasons why machine learning can accommodate very small formats.

2.5.3 A Simplified Floating-Point Format

We now describe a family of simpler formats which allows for cheaper but non-standard operators. The main simplification is that only normal numbers are supported, and we therefore describe this format as Nfloat (for Normal only floats) in this book. As subnormal numbers are not supported, zero (which in IEEE 754 is a particular case of subnormal) cannot be encoded: it must be treated as a special value, like infinities and NaN.

Figure 2.15 describes the $\text{Nfloat}(w_E, w_F)$ format, which is used in the FloPoCo software.

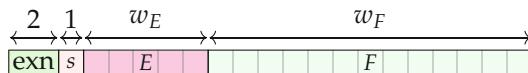


Fig. 2.15 Bit fields of a $\text{Nfloat}(w_E, w_F)$ floating-point number as used in FloPoCo.

A number in this format consists of four fields exn, s , E , and F . The 2-bit field exn encodes the special values explicitly as described in Table 2.3. When $\text{exn} = 01$ the number is normal, and the value encoded is

$$X = (-1)^s \times (1 + F) \times 2^{E - E_0} \quad (2.19)$$

where E_0 is the same exponent bias as in IEEE 754 formats: $E_0 = 2^{w_E-1} - 1$.

Since the extremal values zero and infinity are not encoded as special exponent values, the exponent range for normal numbers is slightly larger than in the IEEE 754 for the same value of w_E , as Table 2.4 shows (also compare Figs. 2.16c and 2.12). However, a number in $\text{Nfloat}(w_E, w_F)$ require two more bits than a number in $\text{IEEEfloat}(w_E, w_F)$.

Table 2.3 Encoding of exceptional cases in the Nfloat format.

exn	Meaning
00	Zero (there are two zeros, denoted as $+0$ and -0 , according to s)
01	Normal numbers
10	Infinity ($+\infty$ or $-\infty$, according to s)
11	NaN (Not a Number)

Table 2.4 Some properties of the IEEEfloat and Nfloat formats.

	$\text{IEEEfloat}(w_E, w_F)$	$\text{Nfloat}(w_E, w_F)$
Bias value		$E_0 = 2^{w_E-1} - 1$
Total size	$w_E + w_F + 1$ bits	$w_E + w_F + 3$ bits
e_{\min}	$-2^{w_E-1} + 2$	$-2^{w_E-1} + 1$
e_{\max}	$2^{w_E-1} - 1$	2^{w_E-1}
Smallest	$2^{e_{\min}-w_F} = 2^{-2^{w_E-1}+2-w_F}$	$2^{e_{\min}} = 2^{-2^{w_E-1}+1}$
Largest	$(2 - 2^{-w_F}) \cdot 2^{e_{\max}}$	$(2 - 2^{-w_F}) \cdot 2^{e_{\max}}$

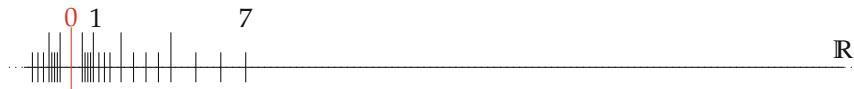
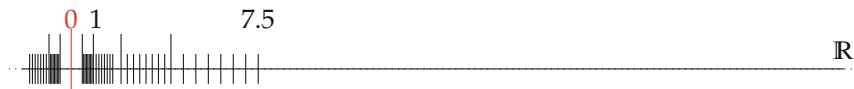
(a) $(w_E, w_F) = (2, 2)$ (b) $(w_E, w_F) = (2, 3)$ (c) $(w_E, w_F) = (3, 2)$

Fig. 2.16 Resolution and range of Nfloat formats when the parameters change: adding one bit to the fraction doubles the number of representable values with the same range. Adding one bit to the exponent increases the range and also reduces the gap around zero.

2.5.4 Respective Motivations of IEEEfloat and Nfloat

The IEEE standard was designed for processor implementations. It is a memory-oriented format designed to make the most out of a fixed number of bits. In particular, exceptional cases are encoded in the two extremal values of the exponent. It also strives to protect the programmer by offering some expected mathematical properties. For example, the main motivation of subnormals numbers is that they bring with them important properties, such as $(x - y = 0) \iff (x = y)$. Indeed, without subnormals (see

Fig. 2.12), this property is not true if one subtracts two numbers close to zero (the difference may be rounded to zero although the numbers are different). See [Mul+18] for more details and other properties that require subnormals.

However, managing these encodings has a cost in terms of performance and resource consumption [EL11]. This is especially true in a hardware pipeline of floating-point operators, where it is clearly more efficient to directly transmit the exception bits to the next operator than to have them encoded then decoded. Subnormal handling has an even higher cost, requiring dedicated shifters, leading zero counters, and MUXes in the operators (this will be studied in details in Chap. 11). This logic can be saved in the absence of subnormals. When designing an application-specific circuit, the application is known at design time, and the format can be matched to it. This reduces the need to “protect the programmer” in arbitrary situations. This is therefore the choice made by FloPoCo and other floating-point libraries.

Besides, one may argue that adding one bit of exponent brings in all the subnormal numbers, and more (compare Figs. 2.16a,c), at a fraction of the cost: subnormals are less relevant if the format is fully parameterized.

This question will probably remain disputable for a long time. In any case, floating-point formats should be fully parameterized to support application-specific arithmetic.

2.5.5 Non-binary Floating-Point Formats

In the two formats studied so far, we have used binary for the mantissa representation as well as for the base of the exponent. These two choices can in principle be generalized to a significand M in some radix β , and an exponent base B , so that a value is represented by a triple (s, M, E) of value $(-1)^s \times M \times B^E$.

This question was thoroughly studied before the standardization of floating point [McK67; BR69; KC73; Cod73; Bre73], and the conclusion was that $B = \beta = 2$, with its implicit bit on normal numbers, provides the best representation efficiency.

2.6 Logarithmic Number Systems

Floating point is not the only way to represent real numbers in hardware circuits with a larger dynamic range than fixed point. One can also use a *logarithmic* coding, or Logarithm Number System (LNS). This book will not cover Logarithm Number System (LNS) in detail. We introduce it here for completeness and also because this book provides many tools that can help a designer implement an LNS system.

2.6.1 An Example LNS Format

Figure 2.17 shows an example of an LNS format, designed to match in range and resolution the FloPoCo format of Fig. 2.15. It is composed of the same two bits for exceptional cases, a sign bit s , and a fixed-point 2's complement representation of a logarithm in base 2, L_X , coded with w_E bits for its integer part and w_F bits for its fractional part. In other words L_X is coded using the sfix($w_E - 1, -w_F$) format.

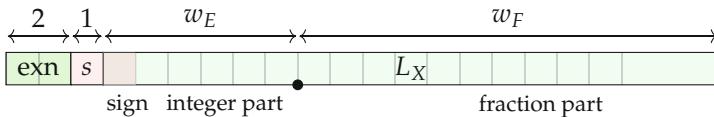


Fig. 2.17 Format of a FloPoCo-like LNS number.

The value represented by the s and L_X fields is

$$X = (-1)^s \times 2^{L_X}. \quad (2.20)$$

As (2.20) cannot represent zero, it has to be treated as a special case. In Fig. 2.17 we chose to encode it in the exn bits.

The other exceptional cases (infinity and NaN) are coded exactly as for floating-point numbers. Exception handling is thus identical in LNS and in floating point. One could choose to encode the exceptional situations in special values of L_X instead: this would allow for a more compact encoding but more expensive decoding. There is no standard here.

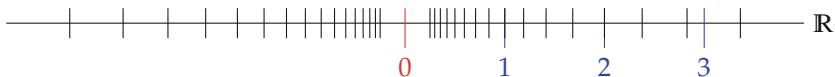


Fig. 2.18 Representation on the real axis of LNS numbers for $(w_E, w_F) = (2, 2)$. Note the gap around zero, similar to the gap of a floating-point format without subnormals. Also note that there are actually two zeroes: $+0$ and -0 .

Figure 2.18 shows the values that can be represented in small LNS formats. Figure 2.19 shows how the set of representable numbers evolves when adding one bit to the integer part or to the fraction part of the format. Again, this is quite comparable to floating point. Comparing these figures to Fig. 2.13 or Fig. 2.12, one may find LNS more elegant since there is no longer the abrupt transition of one binade to the next that can be observed for floating point. However, the issue of the gap around zero remains. Besides, the grouping by binade also has a nice property: we have a full binade of consecutive integers which can be represented exactly in a floating-point

format. In LNS, only powers of 2 are exactly representable (notice that 3 is not representable in Fig. 2.18). For programming languages and applications, having a large range of representable integers is probably much more valuable than having an elegant format. There are many other properties of IEEE floating point that are useful in practice and lost with LNS, for example, the fact that all the representable numbers are rational or the fact that the rounding error in a floating-point addition is always exactly representable as a floating-point number in the same format [Mul+18].

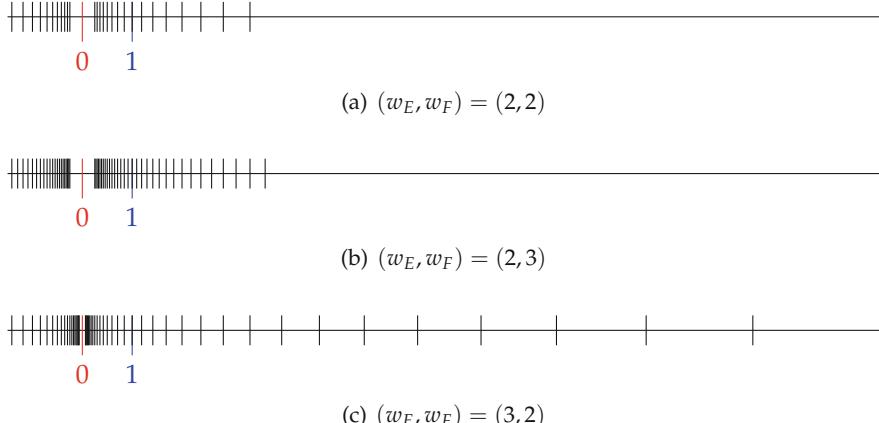


Fig. 2.19 Comparing the resolution and range of LNS formats: adding one bit to the fraction doubles the number of representable values with the same range. Adding one bit to the integer part increases the range and also reduces the gap around zero.

2.6.2 LNS Operations

The main advantage of LNS is the simplicity of multiplication, division, squaring, and square root, implemented, respectively, by addition, subtraction, left shift, and right shift of the logarithms of the operands:

$$L_{X \times Y} = L_X + L_Y, \quad (2.21)$$

$$L_{X / Y} = L_X - L_Y, \quad (2.22)$$

$$L_{X^2} = 2L_X \quad (2.23)$$

$$L_{\sqrt{X}} = \left\lfloor \frac{1}{2}L_X \right\rfloor. \quad (2.24)$$

The main drawback of LNS is that addition and subtraction are much more complicated than in floating point. We briefly overview this here. If X and Y are two positive numbers such that $X > Y$, we have

$$\begin{aligned} L_{X+Y} &= \log_2(2^{L_X} + 2^{L_Y}) \\ &= \log_2(2^{L_X}(1 + 2^{L_Y-L_X})) \\ &= L_X + f_{\oplus}(L_Y - L_X), \quad \text{with } f_{\oplus}(r) = \log_2(1 + 2^r), \\ L_{X-Y} &= \log_2(2^{L_X} - 2^{L_Y}) \\ &= \log_2(2^{L_X}(1 - 2^{L_Y-L_X})) \\ &= L_X + f_{\ominus}(L_Y - L_X), \quad \text{with } f_{\ominus}(r) = \log_2(1 - 2^r). \end{aligned}$$

2.6.3 LNS in Arbitrary Base

So far we have used 2 as the base of the logarithm, but the base can be any positive real b . With $b = 2$, the range and resolution of LNS for a given (w_E, w_F) are comparable to those of floating point with the same parameters. However, the base can be used to fine-tune an LNS format to an application [AGG21]. This is particularly relevant for very small formats. Figure 2.20 illustrates how the choice of b impacts the set of representable numbers.

Even for $b \neq 2$, the logarithm L_X is still encoded as a radix-2 number, essentially to enable efficient binary implementation of (2.21) to (2.24). Note that the base b does not appear in these equations. Conversely, the addition and subtraction functions respectively become $f_{\oplus}(r) = \log_b(1 + b^r)$ and $f_{\ominus}(r) = \log_b(1 - b^r)$.

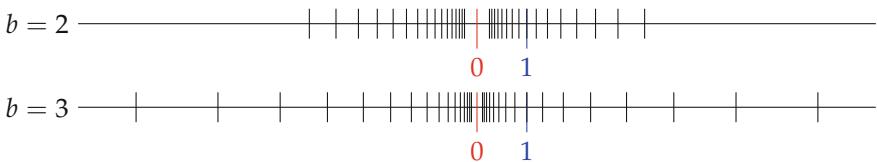


Fig. 2.20 Resolution and range of LNS with different bases, for $(w_E, w_F) = (2, 2)$.

In terms of implementation, base 2 also has the advantage that shifting and leading-zero-count can be used for computing 2^x and \log_2 of integers. Base e (natural logarithm and exponential) has the advantage of simpler Taylor expansions, which may help building simpler hardware for evaluating these functions. For very small formats, all these functions can be tabulated: the choice of base is no longer driven by implementation considerations, but by the range and precision it enables.

In floating point, we could similarly decorrelate the *base* used for the exponent encoding and the *radix* used for the significand fraction, but we would lose the main property on which the floating-point addition is built: multiplication by a power of the base reduces to a shift of digits. As we now understand, this property is lost in LNS whatever the base, so we may as well use any b .

To sum up, the main advantage of LNS is that multiplications, divisions, and square roots are trivial with logarithms. The main drawback is that additions and subtractions are much more complicated. Given this trade-off, there exist applications for which this system is more efficient in terms of speed and area than floating point [Arn+91; Arn+97; CC00; Mat+02; RA03; RA04; KBP13]. In particular, several recent works have investigated the use of small-precision LNS for machine learning [Lee+17; MLM16; Kim+18; Joh18; ACJ20; AGG21]. They will be reviewed in Chap. 24.

References

- [754-08] *IEEE Standard for Floating-Point Arithmetic*. 2008 (cit. on p. 48).
- [754-19] *IEEE Standard for Floating-Point Arithmetic*. also IEEE/ISO/IEC 60559-2020. 2019 (cit. on pp. 7, 330, 332, 357).
- [754-85] *IEEE Standard for Binary Floating-Point Arithmetic*. 1985 (cit. on p. 48).
- [ACJ20] Mark Arnold, Ed Chester, and Corey Johnson. “Training Neural Nets using only an Approximate Tableless LNS ALU”. In: *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2020, pp. 69–72 (cit. on p. 60).
- [AGG21] Syed Asad Alam, James Garland, and David Gregg. “Low-Precision Logarithmic Number Systems: Beyond Base-2”. In: *ACM Transactions on Architecture and Code Optimization* 18.4 (2021), pp. 1–25 (cit. on p. 727).
- [Agr+19] Ankur Agrawal, Silvia M. Mueller, Bruce M. Fleischer, Jungwook Choi, Naigang Wang, Xiao Sun, and Kailash Gopalakrishnan. “DLFloat: A 16-b Floating Point format designed for Deep Learning Training and Inference”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2019, pp. 92–95 (cit. on p. 729).
- [Arn+91] Mark Arnold, Thomas Bailey, J. Cowles, and Jerry Cupal. “Implementing back propagation neural nets with logarithmic arithmetic”. In: *International AMSE Conference on Neural Networks*. 1991 (cit. on p. 60).

- [Arn+97] Mark Arnold, Thomas Bailey, Jerry Cupal, and Mark Winkel. “On the cost effectiveness of logarithmic arithmetic for back-propagation training on SIMD processors”. In: *International Conference on Neural Networks*. Vol. 2. 1997, pp. 933–936 (cit. on p. 60).
- [Avi61] Algirdas Antanas Avižienis. “Signed-Digit Number Representations for Fast Parallel Arithmetic”. In: *IRE Transactions on Electronic Computers* EC-10.3 (1961), pp. 389–400 (cit. on p. 47).
- [BR69] W. S. Brown and P. L. Richman. “The Choice of Base”. In: *Communications of the ACM* 12.10 (Oct. 1969), pp. 560–561 (cit. on p. 56).
- [Bre73] Richard P. Brent. “On the Precision Attainable with Various Floating-Point Number Systems”. In: *IEEE Transactions on Computers* C-22.6 (June 1973), pp. 601–607 (cit. on p. 56).
- [Bur+19] Neil Burgess, Nigel Stephens, Jelena Milanovic, and Konstantinos Monachopolous. “Bfloat16 Processing for Neural Networks”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2019, pp. 88–91 (cit. on p. 729).
- [Car+19] Zachariah Carmichael, Hamed F. Langrudi, Char Khazanov, Jeffrey Lillie, John L. Gustafson, and Dhireesha Kudithipudi. “Performance-Efficiency Trade-off of Low-Precision Numerical Formats in Deep Neural Networks”. In: *Conference for Next Generation Arithmetic*. ACM. 2019, 3:1–3:9 (cit. on p. 53).
- [Cau40] Augustin Cauchy. “Sur les moyens d’éviter les erreurs dans les calculs numériques”. In: *Comptes rendus de l’Académie des Sciences* (1840), pp. 431–442 (cit. on p. 47).
- [CC00] J. N. Coleman and E. I. Chester. “Arithmetic on the European Logarithmic Microprocessor”. In: *IEEE Transactions on Computers* 49.7 (2000), pp. 702–715 (cit. on p. 60).
- [Cod73] William J. Cody. “Static and Dynamic Numerical Characteristics of Floating-Point Arithmetic”. In: *IEEE Transactions on Computers* C-22.6 (June 1973), pp. 598–601 (cit. on p. 56).
- [Dar+20] Bita Darvish Rouhani et al. “Pushing the Limits of Narrow Precision Inferencing at Cloud Scale with Microsoft Floating Point”. In: *Advances in Neural Information Processing Systems*. 2020 (cit. on p. 53).
- [Din+19] Florent de Dinechin, Luc Forget, Jean-Michel Muller, and Yohann Uguen. “Posits: the good, the bad and the ugly”. In: *Conference for Next Generation Arithmetic*. ACM. 2019, p. 6 (cit. on p. 53).
- [Dru+18] Mario Drumond, Tao Lin, Martin Jaggi, and Babak Falsafi. “Training DNNs with Hybrid Block Floating Point”. In: *Advances in Neural Information Processing Systems* 31 (2018) (cit. on p. 53).

- [EL04] Miloš D. Ercegovac and Tomás Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004 (cit. on pp. 3, 11, 23, 214, 218, 259, 267, 303).
- [EL11] Pedro Echeverría and Marisa López-Vallejo. “Customizing floating-point units for FPGAs: Area-performance-standard trade-offs”. In: *Microprocessors and Microsystems* 35.6 (2011), pp. 535–546 (cit. on p. 330).
- [EL04] Miloš D. Ercegovac and Tomás Lang. “On-the-Fly Conversion of Redundant into Conventional Representations”. In: *IEEE Transactions on Computers* C-36.7 (1987), pp. 895–897
- [Erc77] Miloš D. Ercegovac. “A General Hardware-Oriented Method for Evaluation of Functions and Computations in a Digital Computer”. In: *IEEE Transactions on Computers* C-26.7 (1977), pp. 667–680 (cit. on pp. 11, 48).
- [FUD21] Luc Forget, Yohann Uguen, and Florent de Dinechin. “Comparing posit and IEEE-754 hardware cost”. 2021. URL: <https://hal.science/hal-03195756> (cit. on p. 53).
- [Gro18] Posit Working Group. *Posit Standard Documentation*. Release 3.2-draft. 2018 (cit. on p. 53).
- [GT04] Reto Galli and Alexandre F. Tenca. “A Design Methodology for Networks of Online Modules and its Application to the Levinson-Durbin Algorithm”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12.1 (2004), pp. 52–66 (cit. on p. 48).
- [GY17] John L. Gustafson and Isaac T. Yonemoto. “Beating Floating Point at its Own Game: Posit Arithmetic”. In: *Supercomputing Frontiers and Innovations* 4.2 (2017), pp. 71–86 (cit. on pp. 323, 324).
- [HTH19] Greg Henry, Ping Tak Peter Tang, and Alexander Heinecke. “Leveraging the bfloat16 Artificial Intelligence Datatype For Higher-Precision Computations”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2019, pp. 97–98 (cit. on p. 729).
- [Joh18] Jeff Johnson. “Rethinking floating point for deep learning”. 2018. arXiv: 1811.01721v1 (cit. on p. 727).
- [KBP13] I. Kouretas, C. Basetas, and V. Palioras. “Low-Power Logarithmic Number System Addition/Subtraction and Their Impact on Digital Filters”. In: *IEEE Transactions on Computers* 62.11 (2013), pp. 2196–2209 (cit. on p. 60).
- [KC73] H. Kuki and W. J. Cody. “A Statistical Study of the Accuracy of Floating Point Number Systems”. In: *Communications of the ACM* 16.4 (Apr. 1973), pp. 223–230 (cit. on p. 56).
- [Kim+18] Min Soo Kim, Alberto A. Del Barrio, Román Hermida, and Nader Bagherzadeh. “Low-power Implementation of Mitchell’s Approximate Logarithmic Multiplication for Convolutional Neural Networks”. In: *Design Automation Conference (DAC)*. ACM/IEEE, 2018, pp. 617–622 (cit. on p. 727).

- [Lee+17] Edward H Lee, Daisuke Miyashita, Elaina Chai, Boris Murmann, and S. Simon Wong. "Lognet: Energy-efficient neural networks using logarithmic computation". In: *International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2017, pp. 5900–5904 (cit. on p. 727).
- [Li+18] He Li, James J. Davis, John Wickerson, and George A. Constantinides. "Digit Elision for Arbitrary-Accuracy Iterative Computation". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2018, pp. 107–114 (cit. on p. 48).
- [LLH18] Peter Lindstrom, Scott Lloyd, and Jeffrey Hittinger. "Universal coding of the reals: alternatives to IEEE floating point". In: *Conference for Next Generation Arithmetic*. ACM, 2018, p. 5 (cit. on p. 53).
- [Mat+02] R. Matoušek, M. Tichý, Z. Pohl, J. Kadlec, C. Softley, and N. Coleman. "Logarithmic Number System and Floating-Point Arithmetics on FPGA". In: *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2002, pp. 627–636 (cit. on p. 60).
- [McK67] W. M. McKeeman. "Representation error for real numbers in binary computer arithmetic". In: *IEEE Transactions on Electronic Computers* EC-16.5 (Oct. 1967), pp. 682–683 (cit. on p. 56).
- [Mic+22] Paulius Micikevicius, Dusan Stosic, Patrick Judd, John Kamalu, Stuart Oberman, Mohammad Shoeybi, Michael Siu, Hao Wu, Neil Burgess, Sangwon Ha, Richard Grisenthwaite, Naveen Mellemputdi, Marius Cornea, Alexander Heinecke, and Pradeep Dubey. "FP8 Formats for Deep Learning". 2022. arXiv: 2209.05433 (cit. on p. 729).
- [MLM16] Daisuke Miyashita, Edward H. Lee, and Boris Murmann. "Convolutional Neural Networks Using Logarithmic Data Representation". arXiv preprint arXiv:1603.01025. 2016 (cit. on p. 727).
- [Mul+18] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiod, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. 2nd ed. Birkhäuser Boston, 2018 (cit. on pp. 11, 24, 311, 314, 324, 330, 332, 336, 346, 347, 349).
- [Mul94] Jean-Michel Muller. "Some Characterizations of Functions Computable in On-Line Arithmetic". In: *IEEE Transactions on Computers* 43.6 (1994), pp. 752–755 (cit. on p. 48).
- [Nou+22] Badreddine Noune, Philip Jones, Daniel Justus, Dominic Masters, and Carlo Luschi. "8-bit Numerical Formats for Deep Neural Networks". 2022. arXiv: 2206.02915 (cit. on p. 729).
- [NVI20] NVIDIA. *NVIDIA A100 Tensor Core GPU Architecture*. Tech. rep. 2020 (cit. on p. 729).
- [NVI22] NVIDIA. *NVIDIA H100 Tensor Core GPU Architecture*. Tech. rep. 2022 (cit. on p. 53).

- [RA03] J. Ruan and M. Arnold. "Combined LNS Adder/Subtractors for DCT Hardware". In: *Workshop on Embedded Systems for Real-Time Multimedia*. 2003 (cit. on p. 60).
- [RA04] J. Ruan and M. Arnold. "New Cost Function for Motion Estimation in MPEG Encoding Using LNS". In: *International Symposium on Optical Science and Technology (SPIE)*. 2004 (cit. on p. 60).
- [Ska+20] Ali Skaf, Mona Ezzadeen, Mounir Benabdenbi, and Laurent Fesquet. "Clocked and event-driven redundant adjustable precision computing". In: *Microelectronics Reliability* 111 (2020), p. 113729 (cit. on p. 48).
- [Sun+19] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Viji Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. "Hybrid 8-bit Floating Point (HFP8) Training and Inference for Deep Neural Networks". In: *Advances in neural information processing systems* 32 (2019) (cit. on p. 729).
- [Tam+19] Thierry Tambe, En-Yu Yang, Zishen Wan, Yuntian Deng, Vijay Janapa Reddi, Alexander Rush, David Brooks, and Gu-Yeon Wei. "AdaptivFloat: A Floating-point based Data Type for Resilient Deep Learning Inference". 2019. arXiv: 1909.13271 (cit. on p. 729).
- [ZMK22] Sai Qian Zhang, Bradley McDanel, and H.T. Kung. "FAST: DNN Training Under Variable Precision Block Floating Point with Stochastic Rounding". In: *International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 846–860 (cit. on p. 53).

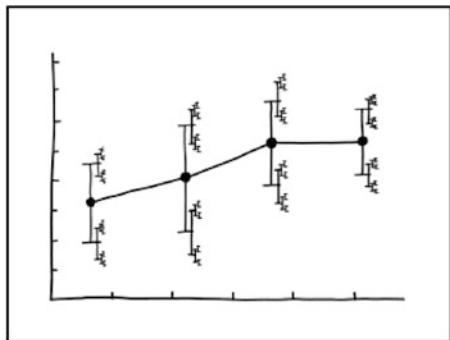
3

CHAPTER 3

Computing Just Right:
Accuracy Specification
and Error Analysis

Reality is frequently inaccurate.

Douglas Adams, *The Restaurant at the End of the Universe*



I DON'T KNOW HOW TO PROPAGATE
ERROR CORRECTLY, SO I JUST PUT
ERROR BARS ON ALL MY ERROR BARS.

Randall Munroe, <https://xkcd.com>

This chapter first defines the accuracy of an operator with respect to the operation it is supposed to implement, discussing in particular various approaches to the notion of rounding. It then states that for efficient application-specific arithmetic, operators should be accurate to their last bit. To reach this objective, it sketches a generic methodology: the error analysis of parametric designs, which can then be exploited to investigate (in an application-specific way) which parameter values lead to best accurate design. This generic methodology is also shown to be well suited to the hierarchical design of complex computing cores assembled out of simpler ones and will be used throughout the book.

Application-specific arithmetic is the art of building non-standard arithmetic operators tailored to the application. By definition these operators lack a standard specification, in particular with respect to the accuracy at which they compute. Still, it is important that accuracy is a first-class concern when building or evaluating operators, on the same level as performance and

hardware cost. Indeed, while it is possible to build fast and cheap operators that compute very inaccurately, one may question the usefulness of doing so. For instance, if an operator computes a 32-bit bit vector that holds only 8 accurate bits and 24 bits of meaningless noise, it is probably interesting to consider outputting the result of this operator on 8 bits only. The operator itself may not be much cheaper nor faster, but it may save hardware and energy further down the computation: why waste energy computing on 24 bits of numerically meaningless data? To avoid this, it seems important to match the accuracy of the operator with its output format.

The purpose of this chapter is to formalize this intuition, first by formalizing the notion of accuracy and then by defining a set of standard notions and practices that allow the designer to construct accurate application-specific operators.

3.1 Accuracy of an Operator

Throughout this chapter, we consider an operator which implements some mathematical operation or function f . We assume for simplicity of presentation that f is a function of $\mathbb{R} \rightarrow \mathbb{R}$, but this discussion can be straightforwardly extended to functions of several variables (including, for instance, the basic operations $+$, $-$, \times , and $/$).

For a given value $x \in \mathbb{R}$, there exists a value $f(x) \in \mathbb{R}$ of the function f , and this is what an operator is supposed to compute (Fig. 3.1, left). An operator for the function will be a piece of hardware that computes a *discrete* version of f , in the sense that the inputs and outputs will be discrete numbers (Fig. 3.1, right).

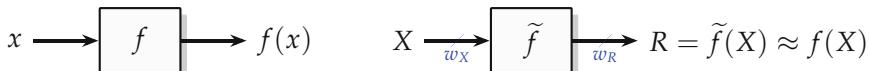


Fig. 3.1 The ideal operator (left) and its implementation (right).

Discretization of the input is not a problem: a discrete number X nevertheless belongs to \mathbb{R} ; therefore the value $f(X)$ is well defined. The problem we address in this chapter is mostly the discretization of the real-valued output value $f(X)$.

The actual operator (Fig. 3.1, right) computes some value $R \approx f(X)$. Hardware so far is still deterministic, so we may define the function computed by the hardware as $\tilde{f}(X)$, and this function is only defined on a discrete subset \mathbb{F} of \mathbb{R} .

3.1.1 Absolute and Relative Error

Definition 3.1 (Absolute Error) The absolute error of the operator with respect to the ideal operation is defined as the difference between the computed value and the ideal one:

$$\delta(X) = \tilde{f}(X) - f(X) .$$

In some contexts, and in particular when the output format is a floating-point one, it is often more useful to define the *relative* error:

Definition 3.2 (Relative Error) The relative error of the operator with respect to the ideal operation is defined as

$$\varepsilon(X) = \frac{\tilde{f}(X) - f(X)}{f(X)} .$$

Note that the error, absolute or relative, is a function of the input X . It can be plotted, for instance (see Fig. 17.14, p. 520, for an example).

In most of this book, we use as a measure of accuracy the *worst-case error bound*, denoted as $\bar{\delta}$ for absolute¹ errors and $\bar{\varepsilon}$ for relative errors:

$$\bar{\delta} = \max_{\forall X} |\delta(X)| ;$$

$$\bar{\varepsilon} = \max_{\forall X} |\varepsilon(X)| .$$

Other measures are possible, such as the average error or, in some digital signal processing contexts, the signal to noise ratio. The worst-case error bound provides the strongest guarantee. It is also relatively easy to compose, as the sequel will show.

Let us now introduce a useful unit of measure for such error bounds.

3.1.2 Unit in the Last Place (ulp)

The notion of unit in the last place (ulp) captures the quantum of discretization of a format. It applies equally well to fixed-point and floating-point formats.

¹ The reader should be aware that the term “absolute” is used here with two different meanings almost in the same sentence: on the one side we have *absolute errors* as opposed to *relative errors*; on the other side the error bound is the max of the *absolute value* of the error (which may be absolute or relative). The authors do apologize for the resulting confusion: renaming either notion seemed preposterous.

Definition 3.3 (Unit in the Last Place in Fixed Point) For a ufix(m, ℓ) or sfix(m, ℓ) format, the ulp is the weight of the least significant bit: $\text{ulp} = 2^\ell$.

The ulp is sometimes denoted as u . It is a constant value for a given fixed-point format. For instance, the ulp of an integer format is 1.

When used as a measure of accuracy, the notion of ulp will allow us to relate the accuracy of an operator to the precision that can be expressed on its output format, without needing to detail this format. For instance, an operator that outputs a ufix(m, ℓ) result with an overall error bound $\bar{\delta}$ smaller than 2^ℓ will simply be said to be accurate to 1 ulp.

The notion of ulp applies equally well to floating-point numbers (the term was actually coined in this context). However, in this case the ulp is no longer a constant of the format: it now represents the weight of the least significant bit (LSB) of the significand and thus depends on the exponent. For an Nfloat(w_F, w_E) format with the notations of the previous chapter, the ulp of a floating-point number X of exponent E_X is $\text{ulp}(X) = 2^{E_X - w_F}$. This function is constant on an interval of the form $[2^k, 2^{k+1})$ (usually called a *binade*, e. g., in Fig. 2.16).

A common practice is to generalize this function $\text{ulp}(X)$ to $X \in \mathbb{R}$. This way, $\text{ulp}(f(X))$ will be defined for the real value $f(X)$, which is not necessarily a floating-point number. The intuition here is that $\text{ulp}(X)$, for $X \in \mathbb{R}$, is the resolution of the format in the neighborhood of X . A subtlety is that this resolution changes at the binade boundaries 2^k (where the exponent changes; see again Fig. 2.16). Several definitions have been proposed for the value that the ulp function should take on such points and compared by Muller [Mul05]. We report here what he concludes is the best definition of the ulp function:

Definition 3.4 (Unit in the Last Place in a Floating-Point Context) If x is a real number that lies between two finite consecutive floating-point numbers a and b , without being equal to one of them, then $\text{ulp}(x) = |b - a|$. Otherwise $\text{ulp}(x)$ is the distance between the two finite floating-point numbers nearest x . Moreover, $\text{ulp}(\text{NaN})$ is NaN .

The “Otherwise” in this definition captures binade boundaries 2^k , but also real numbers larger in magnitude than the largest floating-point number of the format.

3.1.3 Rounding to the Nearest

The output format being defined, the best that an operator can do is output the machine number that is closest to $f(X)$, denoted as $\lfloor f(X) \rceil$. In fixed point, the worst-case absolute error in this case is $\bar{\delta} = \frac{1}{2}\text{ulp}$. For instance, if the output format is sfix($m - 1, 0$), which is a pedantic way of describing

integers on m bits, we have $\text{ulp} = 2^0 = 1$. The value $f(X) = 1.17$ must be rounded to 1 with an error $\delta = 0.17$. The value $f(X) = 1.53$ must be rounded to 2 with an error $\delta = 0.47$. The value $f(X) = 1.49$ must be rounded to 1 with an error $\delta = 0.49$. The worst case is attained for $f(X) = 1.5$ which may be rounded either to 1 or to 2, in both cases with an error $\delta = 0.5 = \frac{1}{2}\text{ulp}$.

If the output format is the fixed-point format $\text{ufix}(-1, -12)$, representing 12-bit numbers in $[0, 1]$, then we have $\text{ulp} = 2^{-12}$, and the worst-case error due to this format is $\text{ulp}/2 = 2^{-13}$, attained for instance when rounding $f(X) = 0.2501220703125 = 0.25 + 2^{-13}$.

If the output is a floating-point number, the value of $\text{ulp}(f(X))$ depends on $f(X)$, but with Definition 3.4, rounding to the nearest still corresponds to an error bound of one half ulp (except when overflowing to infinity).

We notice here that round to the nearest is uniquely defined, except for the values that are exactly in the middle between two consecutive machine numbers. Such values are named *ties*. It is desirable to specify the value to round to in a tie case, so that the rounding to the nearest becomes a well-defined function. For instance, the initial IEEE 754 standard for floating-point arithmetic [754-85] defined the function *round to the nearest, with ties to even* (sometimes abbreviated *round to nearest even*, which is much less clear). Here the phrase *with ties to even* means, in case of a rounding tie, the returned number shall be the one whose significand is even, i. e., has a 0 bit as LSB.

The *ties to even* rule is expensive to implement, because the tie situations must be detected and handled specifically. Classic techniques for this are detailed in textbooks describing the implementation of the IEEE 754 floating-point standard [EL04; Mul+18] and in Chap. 11 of the present book.

When implementing application-specific fixed-point operators, a simpler and cheaper alternative is the following definition:

$$\lfloor Y \rfloor_\ell = \left\lfloor Y + 2^{\ell-1} \right\rfloor_\ell. \quad (3.1)$$

It is simpler to implement, since the floor function $\lfloor \cdot \rfloor_\ell$ simply consists in dropping all the bits of weight strictly smaller than 2^ℓ . The addition adds one single bit, whose value is one half ulp .

This definition always rounds *up* the tie situations, like humans do per convention (e. g., 1.5 will be rounded to 2). This potentially adds a small statistical bias to large computations, which is the reason why the initial IEEE 754 standard decided for a rule where ties are sometimes rounded up and sometimes rounded down.²

Remark that in two's complement, the ties for negative numbers are also rounded up when using (3.1). For instance -1.5 will be rounded to -1 . This

² The cheaper “ties to up” rule was added in the 2008 revision of the IEEE 754 standard [754-08]. It is called *round to nearest, ties to away (from zero)* since what it rounds up is the mantissa fraction, thus rounding positive ties up and negative ties down. However, it remains optional and the *round to nearest, ties to even* mode remains the default.

has to be considered if symmetry around 0 is important. Conversely, the *ties to even* rule does preserve the symmetry.

3.1.4 Truncation Versus Rounding to the Nearest

Note that truncation alone entails a worst-case absolute error bound of 1 ulp. This bound is never exactly attained when truncating a value represented on a finite number of bits: the error is always strictly smaller than 1 ulp. This is in contrast to the half-ulp error bound of rounding to the nearest, which is attained in the tie situations.

We now see a first trade-off related to rounding. Consider the case when we can compute $f(X)$ exactly, but it fits on more bits than the output format can hold: we have to round it. One option is to simply truncate it, which is for free in hardware (some bits are just dropped), with a worst-case error bound of 1 ulp. Another option is to round it to the nearest using (3.1). It has a smaller error ($\frac{1}{2}$ ulp instead of 1 ulp), but it is not for free (the addition of $2^{\ell-1}$ may entail a carry propagation on all the bits of the output and therefore requires an adder).

Our purpose at this point is only to expose this trade-off. Which alternative is the best really depends on the context, and this book will present several instances where one solution is clearly better than the other. In particular, a common trick is to add the value $2^{\ell-1}$ (sometimes called *rounding bit*, or *rounding half-ulp*) for free in some previous part of the internal computation. Then rounding to the nearest becomes cheap as truncation and will be preferred without discussion. This trick will be used several times in this book.

Another point of view is to consider that we have two ways to achieve an error bound of $2^{\ell-1}$. One is to round to the nearest to a format whose ulp is 2^ℓ . The second is to truncate to a 1-bit larger format, whose ulp is $2^{\ell-1}$. There is still a trade-off: the first solution costs one more addition (unless the previous trick can be used). The second is for free, but provides the result on a 1-bit larger format, which will usually entail some other overhead in subsequent computations.

3.1.5 Rounding and Overflow Specification in Fixed-Point Libraries

Most standard fixed-point libraries, such as `fixed_pkg` for VHDL or `ap_fixed` and `ac_fixed` for C++ high-level synthesis (HLS), offer the possibility to specify a rounding mode. The choice always include round to

the nearest and round toward $-\infty$ (truncation). The reader should now be aware of the trade-off involved: round to the nearest is more accurate than truncation, but may entail additional hardware cost due to the addition of the rounding half-ulp.

Libraries also sometimes offer other directed rounding modes that, in fixed point, are not implemented by truncation: round toward zero and round away from zero for a two's complement number, round toward $+\infty$ for both signed and unsigned numbers. These have the same one-ulp worst-case error as truncation, but they are more expensive to implement. Actually they are even more expensive than round to nearest with ties to up: on top of the rounding addition, they need to manage the special case when all the bits to be rounded off are zeroes (e.g., 17.01 should be rounded up to 18, but 17.00 should be rounded up to 17). Additional hardware is needed to detect this case. Thus, round to the nearest will be at the same time cheaper and more accurate. These directed rounding modes should therefore only be used for a good reason, e.g., to compute safe bounds of a value using interval analysis [MKC09].

Note that these packages also allow you to specify the behavior in case of overflow: it can be a truncation of the overflow bits (sometimes called “wrap” since it has the effect of wrapping around large values to small ones; see Fig. 2.2), or saturation (sometimes called “clipping”), which maps all positively/negatively overflowing values to the largest/smallest representable one, respectively. Again, the reader should be aware of the trade-off: truncation costs nothing in hardware, but may be numerically catastrophic. Conversely, saturation is numerically better, in particular in signal processing applications, but it costs at least one multiplexer (in contrast saturated arithmetic is for free in a software context, when supported by the instruction set of the processor). In application-specific circuits, we have the option to implement saturation for all the operations, but we also have the option of using a wider, overflow-free internal datapath and saturating only once at the end. The first option will be less accurate and sometimes also more expensive.

In this book, we tend to use range and error analysis to avoid overflow by construction; therefore, there will be very few mentions of this issue, but this conservative approach is not possible in all situations.

3.1.6 Half-Unit Biased (HUB) Format

The half-unit biased (HUB) format is a variant of fixed-point format that has been used implicitly in several publications [DM95; SS99; DT05], then defined formally by Hormigo and Villalba [HV16b] in the context of application-specific architectures [HV14; MH15] and as a basis for a floating-point representation [HV16b; HV16a].

The HUB format simply adds an implicit constant half-ulp to a fixed-point representation, as illustrated by the example provided in Table 3.1.

Table 3.1 The signed HUB(1,−2) representation. The implicit half-ulp is in bold.

HUB code	Represented binary value	Represented decimal value
0.11	0.1 11	0.875
0.10	0.10 1	0.625
0.01	0.01 11	0.375
0.00	0.00 11	0.125
1.11	1.1 11	−0.125
1.10	1.10 1	−0.375
1.01	1.01 11	−0.625
1.00	1.00 11	−0.875

As shown by this table, HUB has one big drawback: zero is not represented exactly. It has, however, two main advantages.

The first is that the set of representable HUB numbers is symmetrical around zero, and the opposite of a number is computed by bit-wise inversion. This is perfectly illustrated by Table 3.1. In other words, computing the opposite is cheaper than in two's complement, where a carry propagation is required in addition to this bitwise inversion.

The second advantage is that the rounding of a binary number (be it HUB or classic two's complement) to the nearest HUB number is always simply by truncation, hence for free. This property is best illustrated in Table 3.2 by a few examples of rounding values close to 0.5 to the HUB format already used in Table 3.1.

Table 3.2 Example of rounding to the nearest signed HUB(1,−2): the error bound is still one half-ulp, here $2^{-3} = 0.125$.

x		$\lfloor x \rfloor$			
Binary	Decimal	HUB	Decimal	Rounding error	
0.1000000...	0.5	0.10	0.625	-2^{-3}	
0.100001	$0.5 + 2^{-6}$	0.10	0.625	$-2^{-3} + 2^{-6}$	
0.101111	$0.75 - 2^{-6}$	0.10	0.625	$2^{-3} - 2^{-6}$	
0.101111...	$0.75 - \varepsilon$	0.10	0.625	$2^{-3} - \varepsilon$	

In short, the HUB format enables half-ulp accuracy at zero cost (only truncation), thus providing an elegant solution to the dilemma exposed in the previous section.

In addition, as illustrated by the two middle lines of Table 3.2, when rounding (by truncation) a HUB(m, ℓ_1) value to a HUB(m, ℓ_2) value, with $\ell_2 > \ell_1$, the half-ulp error bound is never attained: the maximum absolute error in this case is $\delta = 2^{\ell_2-1} - 2^{\ell_1-1}$. This is a consequence of the symmetry of the set of representable numbers.

It has been suggested to build a full computing system using HUB formats [HV16b; HV16a]. However, in the present book, we are more conservative: we will only be using HUB as an intermediate, internal format when it improves the cost/accuracy trade-off. Examples include FIR filters [HV14], CORDIC rotators [MH15], or the multipartite method for function approximation [DT05] presented in detail in Chap. 17.

3.1.7 Rounding to the Nearest When an Approximation Is Involved

A common situation is that the computation of $f(X)$ is actually performed using some approximation. This will be the case for most transcendental functions, but for simplicity we take a simple example: multiplication by π (itself a transcendental number). The function to evaluate is $f(X) = \pi X$. Assume we want the input format to be some ufix($-1, \ell$) ($\ell < -1$), so $0 \leq X < 1$. Also assume we want the output format to have the same precision ℓ . The value taken by the output is bounded by $\pi < 4$; therefore the output format will be ufix($1, \ell$).

In this toy example, the issue is that the binary representation of π is an infinite sequence of seemingly random bits. For this reason, we cannot simply compute the exact result and then round it: computing the exact result would require (in the general case) an infinite amount of computation.

The natural idea here is to use, for this computation, an *approximation* $\tilde{\pi}$ to π that is good enough. We will soon refine what we mean by “good enough”; for now, the main point is that we have to implement an approximate computation, because the exact computation is not possible.

For instance, let us try to use an approximation of π with precision identical to the output precision:

$$\tilde{\pi} = \lfloor \pi \rceil_\ell.$$

This entails an *approximation error* that we can define as

$$\delta_\pi = \tilde{\pi} - \pi \tag{3.2}$$

which is bounded by

$$|\delta_\pi| \leq 2^{\ell-1}. \tag{3.3}$$

Now we may build a first finite architecture that computes an approximation of $f(X) = \pi X$: it simply consists of a multiplier inputting X and $\tilde{\pi}$ (left part of Fig. 3.2) followed by a rounding (right part of Fig. 3.2). We can define and bound the error of the result of the multiplier with respect to the function we intend to compute:

$$\begin{aligned}\delta_{\text{approx}}(X) &= \tilde{\pi}X - \pi X = (\tilde{\pi} - \pi)X \\ |\delta_{\text{approx}}(X)| &= |\tilde{\pi} - \pi| \cdot |X| \leq 2^{\ell-1}.\end{aligned}\quad (3.4)$$

The simplest option is to use an exact fixed-point multiplier, because it does not add an error term. However, this multiplier will compute, among others, bits of weight $2^{2\ell}$ (the products of the bits of weights 2^ℓ of both inputs). Therefore, the result of this multiplier will not be in the target format: it must be rounded. This entails a second rounding error that we can define and bound as

$$\delta_{\text{round}}(X) = |R(X) - \tilde{\pi}X|,\quad (3.5)$$

where $R(X)$ is the rounded result. Thus, a naive rounding to the output precision ℓ results in

$$\delta_{\text{round}}(X) < 2^{\ell-1}.\quad (3.6)$$

The overall error of the computation becomes

$$\begin{aligned}\delta(X) &= R(X) - \pi X \\ &= R(X) - \tilde{\pi}X + \tilde{\pi}X - \pi X \\ &= \delta_{\text{round}}(X) + \delta_{\text{approx}}(X).\end{aligned}$$

Using the triangle inequality, $|\delta(X)| < 2^{\ell-1} + 2^{\ell-1} = 2^\ell$. We observe that this technique does not achieve rounding to the nearest, which would correspond to a bound $|\delta(X)| < 2^{\ell-1}$.

Is it possible to build a finite architecture that computes this function correctly rounded to the nearest? The answer is yes, but it will (1) require a finer error analysis and (2) probably be more expensive.

First, let us convince the reader that a finite architecture is possible. There is only a finite number of possible values of the fixed-point input X , so we could, for instance, tabulate all the products by all the possible inputs, each correctly rounded to the nearest. Of course, we have only delegated the problem to the software that must compute these values. In this particu-

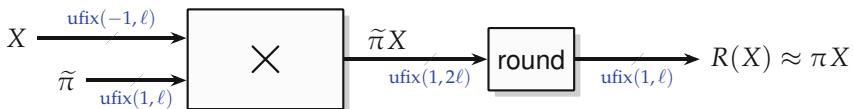


Fig. 3.2 An architecture approximating multiplication by π .

lar case, each correctly rounded product can be precomputed in finite time by an algorithm that uses more and more precision for $\tilde{\pi}$ until the precision is sufficient to decide rounding. Such an algorithm terminates, precisely because π is irrational: none of the values of πX is a rational; therefore, none of the values of πX can be exactly halfway between two consecutive fixed-point numbers since such half-way points are rationals (of the form $\frac{n}{2^{\ell-1}}$ for some integer n).

However, to achieve rounding to the nearest using an architecture similar to Fig. 3.2, we definitely need to refine the error analysis. One problem is the error described by (3.6): as long as the final “round” box entails at most one half-ulp of error, whatever tiny approximation error added to this final round error will sum to an overall error larger than the half-ulp, preventing rounding to the nearest. Note that this half-ulp error bound is attained each time the intermediate result $\tilde{\pi}X$ is exactly halfway between two consecutive ufix($1, 2\ell$) numbers.

The same issue appears in the rounding of the constant as described by (3.3). Thus, we can play with the precision we use for $\tilde{\pi}$: the architecture can use more bits of π . For instance, we can use the largest precision that was ever needed by the previous table-filling algorithm. Indeed, doing so ensures that the intermediate product will never be a half-way case; hence, it restores some hope to achieve rounding to the nearest. However, obviously, it will entail a larger multiplier.

We stop this discussion here: our purpose is not to show how to build constant multipliers rounded to the nearest (this issue [BMR04; BDM08; Din+19; GVK22] will be surveyed in Chap. 12). The purpose of this example was to illustrate the following very general result:

As soon as a computation involves some approximation, achieving an 1-ulp absolute error bound is much easier than achieving rounding to the nearest. In the general case, there are more error terms than in our toy example. In particular there may be many intermediate roundings and several layers of approximation. The value of all these error terms can, in general, be made arbitrarily small by computing with internal precisions that are arbitrarily larger than the output precision. However, just the final rounding of the intermediate result will add one half ulp to the overall error of the operator. This can make rounding to the nearest, which corresponds to an error bound of one half ulp, very challenging.

Note that round to nearest remains relatively easy to achieve for simple operations: indeed, the IEEE 754 standard mandates that operators should return round to nearest for addition, subtraction, multiplication, division, and square root. For instance, for multiplication, if the second input is no longer the irrational π , but a finite fixed-point or floating-point number, then the error term δ_{approx} disappears: the only remaining error is the final round error, which is indeed bounded by an half-ulp. For division, it may seem less intuitive because the binary representation of a quotient like $1/3$ is infinite (as it is in decimal). However, this infinite representation is periodic and

can therefore be computed in finite time. For instance, the paper-and-pencil algorithm computes a quotient and a remainder, and the remainder can be used to determine the rounding to the nearest of the quotient. The details are found in textbooks [EL94; Kor02; EL04; Par10] and will be presented in Chap. 9 of the present book.

Conversely, for some numerical functions, there currently exists no algorithm to compute the rounding to the nearest that is proven to terminate on all inputs. This is called the Table Maker’s Dilemma, and the interested reader is encouraged to read Muller’s textbook for an extensive presentation and the state of the art on the subject [Mul16]. For an intuition of why this problem is difficult, consider that we needed the fact that π is transcendental to prove that our algorithm for πX would eventually terminate. For many classic special functions, there is a lack of a similar number-theoretic result, and indeed the current software implementations are either not rounded to the nearest or not proven to terminate.

3.1.8 Faithful Operators and Last-Bit Accuracy

When rounding to the nearest is prohibitively expensive to achieve, the operators in this book will be built with the relaxed constraint of a 1-ulp error bound.

Definition 3.5 (Last-Bit Accuracy) An operator is said to be faithful, or equivalently last-bit accurate, when its absolute error bound is strictly smaller than the ulp of its output, or

$$\bar{\delta} < u \quad . \tag{3.7}$$

Considering that the output format implies for the error bound $\bar{\delta}$ that $\bar{\delta} \geq u/2$, it is still a tight specification. In particular, it has the following properties, illustrated by Fig. 3.3:

- A faithful operator will return one of the two machine-representable value closest to the exact value $f(X)$.
- If the exact value $f(X)$ happens to be a machine-representable number, then a faithful operator will return exactly this value. Note that for this property to hold, the error bound must be *strictly* smaller than 1 ulp.

This is sometimes called *faithful rounding* in the literature (and in the present book), but some authors argue that a rounding should be a function, and the faithful accuracy constraint does not translate to a rounding function: as Fig. 3.3, left, illustrates, there are sometimes two possible output values for a faithful operator.

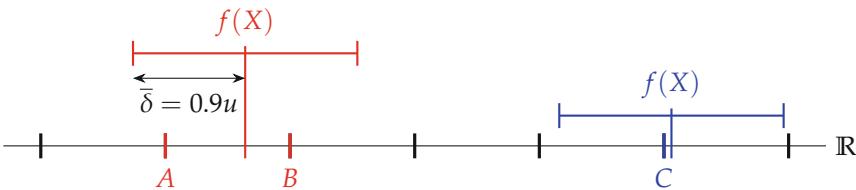


Fig. 3.3 Representation on the real axis of the machine-representable numbers in some discrete format and two cases of faithful rounding with $\bar{\delta} = 0.9u$: left, the operator may return either A or B ; right, the operator may only return C .

3.1.9 Last-Bit Accuracy Versus Rounding to the Nearest

We can now introduce a second trade-off a designer must face when designing application-specific arithmetic. For the same application-level accuracy $\bar{\delta}_{\text{target}} = 2^k$, a designer must choose between:

- an operator that is 1-ulp accurate for a format whose ulp is 2^k ,
- or, an operator that computes rounding to the nearest (which is more expensive) for a format whose ulp is 2^{k+1} (this format is one bit smaller).

In most of the cases, choosing the faithful option is cheaper. Even for multiplication, it is possible to build faithful truncated multipliers (reviewed in Chap. 8) that save significant resources. However, the wider output format will probably entail some overhead in the operators that consume this output, so the best choice actually depends on the context.

3.2 Using the Format to Specify the Accuracy

We can now come back to a question asked in the introduction of this chapter: does it make sense to develop architectures that are worse than faithful?

It is difficult to be really conclusive here. It certainly makes no sense to output many bits that have no numerical meaning. However, the 1 ulp bound of faithful accuracy is quite arbitrary and chosen for simplicity. A bound of 0.75 ulp would provide slightly better application-level accuracy while being slightly more expensive to achieve. It would not fundamentally require different techniques. We hope to have conveyed that the bound 0.5 ulp is the one that is fundamentally more difficult.

Conversely, there are certainly situations where, due to a threshold effect, an operator accurate to 1.5 ulp will be a better choice than one accurate to 1 ulp. If this operator is hidden within a larger application and if the 1.5-ulp error is taken into account in the global error analysis, there is really nothing wrong with that.

One good reason to standardize on one-ulp accuracy is that it considerably simplifies the interface to an operator generator: once the user has specified the format, the ulp is defined; hence the target accuracy is specified as well. Most operators in the FloPoCo software follow this convention, because it removes the need to specify their accuracy. This removes one degree of freedom from the designer, but (as a designer) the author of these lines believes this degree of freedom was a burden. It also eases design-space exploration by having a clear, shared accuracy specification: we may compare architectures that have the same accuracy, hence are numerically equivalent, and select the best one according to the other criteria (cost and performance).

3.3 Designing Accurate Operators

To conclude this chapter, here are a few guidelines that are followed throughout this book to design accurate operators, for instance, faithful ones.

We illustrate these guidelines on the following example: the evaluation of the logarithm function using the Taylor series³ on some interval centered on 1, which corresponds to

$$\log(1 + x) \approx x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \dots$$

This operator will return, for each input X , a value $\tilde{f}(X)$. While the overall error of the operator is easy to define as $\delta(X) = \tilde{f}(X) - f(X)$, there are many contributions to this error. We can list:

- the approximation error due to limiting the series to some degree,
- the fact that we have to round some coefficients, for instance, $1/3$, to some precision,
- the rounding that will happen in some intermediate operations,
- a final rounding, as we had in the example of multiplication by π .

In this section, we address three problems. The first is that there are many *internal parameters* hidden in the above list (“some degree,” “some precision,” etc.). Besides that, there are architectural design choices, such as the evaluation order of the polynomial, which may impact cost, performance, and accuracy. We also consider such choices as internal parameters of the architecture. Section 3.3.1 advocates to keep all these choices open as late as possible by designing parametric architectures.

³ The reader should be aware that this is again a toy example. Taylor series will actually be used in some cases in this book, but in general there are much better polynomial approximation techniques. They will be reviewed in Chap. 18.

The second problem is to be able to compute formally the accumulation of all these sources of error, in order to bound it. This is usually called *error analysis*, and a general method for this is discussed in Sect. 3.3.2. It should be parametric as well, and formal enough to be implemented as a program that inputs the various architectural parameters and outputs the error value.

The third problem is to *optimize* the resulting architecture, i. e., finally decide the best value for all the internal parameters. The general philosophy here is “computing just right”: we want to be sure that each sub-component (each multiplier in our polynomial architecture) is accurate enough, but we also want to avoid components that are uselessly accurate. Section 3.3.3 advocates design-space exploration algorithms that take into account performance and cost, but also accuracy thanks to parametric error analysis.

3.3.1 Parametric Designs

An operator can be parametric in two (fuzzily defined) ways. First, *external* parameters allow the operator to interface to a variety of environments. They are exposed to the user as knobs for tuning the operator to its context. The most obvious external parameters, for instance, describe the input and output formats, but an operator can also have parameters that allow the user to control some trade-offs in the implementation, for instance, cost versus performance.

Such a parametric architecture typically also has *internal* parameters whose value is deduced from the external parameters. For instance, in a floating-point adder parameterized by a significand size of w_F , some of the internal signals may be of size $w_F + 1$ or $w_F + 3$ or $\lceil \log_2(w_F + 2) \rceil$. The full architecture becomes parametric.

What we advocate here is the following: once the discipline is in place to write parametric architectures, a designer should not be afraid to add even more parameters. When faced with an architectural design choice, it is a very productive strategy to delay the choice by capturing it as a new internal parameter. Doing so has one single drawback (it makes the architecture code generation slightly more complex), but provides many advantages:

- It leads to a much finer understanding of the relation between several design choices.
- It clearly separates the problem of generating the architecture from the problem of making architectural design choices. The complexity of the former is hardware and technical. The complexity of the latter is software and algorithmic.
- It enables design space exploration, hence architecture optimization.
- Operators are much more future proof, because a design choice can be reassessed easily in the light of evolutions of the targeted technology.

Many publications present an architecture that is only one instance in a world of possible operators. In such cases, adding the missing parameters is the guarantee to implement a solution that is at least as good as the published one, since it captures it, but also potentially better alternatives.

3.3.2 Error Analysis

While it may be useful in the early design phase to think in vague terms such as “the error of this term is about 4 ulps,” a formal and automated error analysis requires much more care: there is no such thing as “the error of a term.”

The first principle of error analysis is that **an error is a difference (absolute or relative) between a less accurate value and a more accurate value**. The difference precisely captures by how much one value is more accurate than the other.

Our definition of the overall error from Sect. 3.1 follows this convention: $\delta(X) = \tilde{f}(X) - f(X)$. Here $f(X)$ is the ideal mathematical value and is more accurate than $\tilde{f}(X)$, the value computed by the operator. We have already seen several other instances of such definitions, in Eqs. (3.2)–(3.6). The first step of error analysis is to properly define all these errors, attempting to capture each and every source of error.

The values used in the error analysis may be purely virtual, in the sense that they never actually appear in the architecture. For instance, consider the approximation error between our example $\log(1 + X)$ and the Taylor series truncated to degree d : we must define the virtual value

$$p(X) = X - \frac{X^2}{2} + \frac{X^3}{3} \dots + (-1)^{d+1} \frac{X^d}{d}$$

so that we may formally define the two errors

$$\delta_{\text{approx}}(X) = p(X) - \log(1 + X)$$

$$\varepsilon_{\text{approx}}(X) = \frac{p(X) - \log(1 + X)}{\log(1 + X)} .$$

Here the value of the polynomial is a virtual one; it is a purely mathematical object just like $\log(1 + X)$. For most inputs X , its binary representation would still be infinite, for instance, due to the division by 3. It is only used as an intermediate value in the error analysis.

Adhering to this first principle allows us to compose errors as follows. If A approximates B and B itself approximates C , then it is easy to capture how A approximates C using the two identities:

$$A - C = (A - B) + (B - C) \quad (3.8)$$

$$\frac{A - C}{C} = \frac{A - B}{B} + \frac{B - C}{C} + \frac{A - B}{B} \times \frac{B - C}{C} \quad (3.9)$$

or, in terms of errors,

$$\delta_{AC} = \delta_{AB} + \delta_{BC} \quad (3.10)$$

$$\epsilon_{AC} = \epsilon_{AB} + \epsilon_{BC} + \epsilon_{AB} \cdot \epsilon_{BC}. \quad (3.11)$$

Of course these identities can be decomposed further. For instance, if it turns out that there are several rounding errors involved in the approximation of B by A , then we must try to isolate them one by one by defining more virtual values. Let us illustrate this on our logarithm example. We may use (3.8) to isolate the Taylor approximation error as follows:

$$\delta(X) = (\tilde{f}(X) - p(X)) + (p(X) - \log(1 + X)).$$

Here, the term $p(X) - \log(1 + X)$ can be bounded thanks to mathematical results about Taylor series (the bound will depend on the width of the interval): we no longer worry about it. The remaining term, $\tilde{f}(X) - p(X)$, captures the combination of all the rounding errors that we have to make when moving from the reals to the discrete numbers of the architecture. But this term is still too complex for us to bound it. Therefore, we must attempt to split it to isolate the source of each error one by one.

Let us just perform one step of this task for illustration. We may define an intermediate virtual value $p_2(X)$ that just replaces in $p(X)$ the value $\frac{X^2}{2}$ by the value $T_2(X)$ that is actually computed by the architecture:

$$p_2(X) = X - T_2(X) + \frac{X^3}{3} \dots + (-1)^{d+1} \frac{X^d}{d}.$$

Here $p_2(X)$ is still a virtual value, but we have progressed toward the architecture: the term $T_2(X)$ is a concrete value; it corresponds to a bit vector in the architecture.

Assume that we have on our shelf (or that we decide to build for the occasion) a parametric operator that can compute a faithful approximation to $\frac{X^2}{2}$: it will verify

$$\left| T_2(X) - \frac{X^2}{2} \right| < 2^{\ell_2}.$$

Here ℓ_2 is the LSB of $T_2(X)$: it is one of the internal parameters of the architecture we are building. We do not fix its value at this point, we just leave it as a parameter.

Now, if we compute the error between $p_2(X)$ and $p(X)$, we find

$$p_2(X) - p(X) = T_2(X) - \frac{X^2}{2}$$

$$|p_2(X) - p(X)| < 2^{\ell_2}.$$

We now have a longer chain of successive approximations

$$\delta(X) = \underbrace{(\tilde{f}(X) - p_2(X))}_{\text{bounded by } 2^{\ell_2}} + \underbrace{(p_2(X) - p(X))}_{\text{bounded by Taylor}} + \underbrace{(p(X) - \log(1+X))}_{\text{bounded by Taylor}}$$

in which we know how to bound the second and third error terms. It remains to bound the first term $\tilde{f}(X) - p_2(X)$, which can be achieved by inserting more intermediate approximate values that capture the rounding errors due to the other operations of the polynomial.

The main purpose of error analysis is to build such chains of successive approximation values, some virtual, some not.

Then, obtaining a bound on the absolute value of the error simply requires to use the triangle inequality: if

$$\delta(X) = \delta_1(X) + \delta_2(X) + \cdots + \delta_k(X),$$

then

$$|\delta(X)| \leq |\delta_1(X)| + |\delta_2(X)| + \cdots + |\delta_k(X)|,$$

hence

$$|\delta(X)| \leq \bar{\delta} = \bar{\delta}_1 + \bar{\delta}_2 + \cdots + \bar{\delta}_k.$$

Of course, the composition of errors is sometimes more tricky than in the above example (which was carefully chosen to involve only additions of error terms). In particular, when computing the error bound for a product, one must take into account the ranges of the values being multiplied, not only the error terms to which they participate. However, this task will remain tractable as long as we stick to the principles summarized below:

- define an error as a difference between a less accurate term and a more accurate one,
- introduce as many virtual values as needed,
- introduce all the concrete values that appear in the architecture,
- introduce architecture parameters in the process,
- bound the errors as late as possible, so that the error computation can use algebraic simplifications.

For the designer who wants to be sure that no error term was forgotten, this methodology lends itself quite well to formal proof. In particular, the Gappa formal proof assistant⁴ was designed for this purpose [DLM11].

⁴ <https://gappa.gitlabpages.inria.fr>.

Gappa will perform the tedious task of bounding the combination of error terms in a very safe way. It supports floating-point formats as well as fixed-point formats. Its only drawback is that it is not well suited to parametric design: a Gappa proof script must be written for a given instance of the parameter values. However, this is relatively easy to automate.

3.3.3 Error Budget and Design-Space Exploration

At this point, we are able to build an error analysis that computes the error of an architecture out of its internal parameters. We now show how this knowledge may be converted to an operator computing just right, i. e., faithful (or more generally guaranteeing a prescribed accuracy) and at the same time not uselessly accurate.

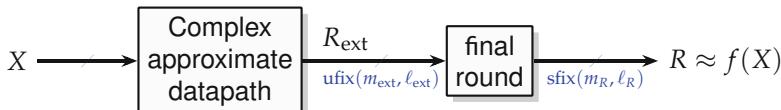


Fig. 3.4 The generic architecture of a faithful operator.

First, the generic architecture to envision is most probably the one of Fig. 3.4. It will internally compute an extended-precision intermediate result R_{ext} . Its precision is extended with respect to the output format: in most cases we will have $\ell_{\text{ext}} < \ell_R$. However, this internal result does *not* need to be faithful to the internal precision ℓ_{ext} . Indeed, the error of this architecture can be decomposed as follows:

$$R - f(X) = \underbrace{(R - R_{\text{ext}})}_{=\delta_{\text{final round}}} + \underbrace{(R_{\text{ext}} - f(X))}_{=\delta_{\text{datapath}}}, \quad (3.12)$$

where

- $\delta_{\text{final round}} = R - R_{\text{ext}}$ is the error due to the final rounding in the **final round** box. It is incompressible,⁵ bounded by $u/2$.
- $\delta_{\text{datapath}} = R_{\text{ext}} - f(X)$ is the overall error of the main datapath. To ensure a faithful operator, this error must remain strictly smaller than $u/2$.

Our task is therefore to make sure that the datapath error δ_{datapath} , which includes the sum of all the approximation and rounding errors, remains strictly smaller than $u/2$.

⁵ Actually, using internally the HUB format can sometimes reduce this final rounding error to $u/2 - 2^{\ell_{\text{ext}}-1}$. This trick is probably due to Das Sarma and Matula [DM95; DT05] and is demonstrated in Sect. 17.2 of this book.

This is not difficult in practice:

- Any approximation technique has one or more parameters that control the accuracy of the approximation. For the approximation to π , it was simply the number of bits of π considered. For the Taylor approximation, it was the degree d . For other iterative algorithms, it will be the number of iterations.
- All the rounding errors can be made arbitrary small by increasing the internal precision used. The extra bits added internally to the output precision are often called *guard bits*. For instance, on Fig. 3.4, R_{ext} is computed with $g = \ell_R - \ell_{\text{ext}}$ guard bits.

In other terms, it is not difficult to find a combination of the architectural parameters that fulfills the constraint that the corresponding overall error is smaller than $u/2$.

Finding the *optimal* combination is a different story, and it remains an art. It is complex because the objective is to minimize some cost (e.g., the area, the delay, or the power consumption of the architecture), which usually depends on the parameters in a non-trivial way. In some cases, the problem can be expressed in a generic optimization framework, such as integer linear programming (ILP) (see Appendix B). In some other cases, an ad hoc method can be used, because there are not too many parameters and for some of them, exhaustive enumeration of their values is possible. The following chapters will present many such optimization problems.

3.4 Now Go Divide and Conquer

To conclude this chapter, note that the accuracy-centered design method presented here allows the divide-and-conquer construction of very complex operators. For instance, in our logarithm case study, we have assumed the existence of a parametric faithful operator for computing $X^2/2$. To build this sub-component (which is a simpler function), the same methodology can be used.

The most complex example of such a divide-and-conquer approach in this book is, in Chap. 22, a parametric faithful floating-point exponential unit (computing e^x) that is built out of many subcomponents including faithful constant multipliers, a faithful polynomial approximator (itself built out of faithful truncated multipliers), and tables of precomputed values. Note that a variant of this operator has been used to build an even more complex one for x^y [Din+13].

References

- [754-08] *IEEE Standard for Floating-Point Arithmetic*. 2008. (cit. on p. 69).
- [754-85] *IEEE Standard for Binary Floating-Point Arithmetic*. 1985. (cit. on p. 69).
- [BDM08] Nicolas Brisebarre, Florent de Dinechin, and Jean-Michel Muller. “Integer and Floating-Point Constant Multipliers for FPGAs”. In: *International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE, 2008, pp. 239– 244. (cit. on p. 75).
- [BMR04] Nicolas Brisebarre, Jean-Michel Muller, and Saurabh Kumar Raina. “Accelerating Correctly Rounded Floating-Point Division when the Divisor Is Known in Advance”. In: *IEEE Transactions on Computers* 53.8 (2004), pp. 1069–1072. (cit. on p. 75).
- [Din+13] Florent de Dinechin, Pedro Echeverría, Marisa López-Vallejo, and Bogdan Pasca. “Floating-Point Exponentiation Units for Reconfigurable Computing”. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 6.1 (2013). (cit. on p. 84).
- [Din+19] Florent de Dinechin, Silviu-Ioan Filip, Luc Forget, and Martin Kumm. “Table-Based versus Shift-And-Add Constant Multipliers for FPGAs”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2019. (cit. on p. 75).
- [DLM11] Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond. “Certifying the floating-point implementation of an elementary function using Gappa”. In: *IEEE Transactions on Computers* 60.2 (2011), pp. 242–253. (cit. on p. 82).
- [DM95] Debjit Das Sarma and David W. Matula. “Faithful Bipartite ROM Reciprocal Tables”. In: *12th Symposium on Computer Arithmetic*. Ed. by S. Knowles and W.H. McAllister. IEEE, 1995, pp. 17–28. (cit. on pp. 71, 83).
- [DT05] Florent de Dinechin and Arnaud Tisserand. “Multipartite Table Methods”. In: *IEEE Transactions on Computers* 54.3 (2005), pp. 319–330. (cit. on pp. 71, 73, 83).
- [EL04] Miloš D. Ercegovac and Tomás Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004. (cit. on pp. 69, 76).
- [EL94] Miloš D. Ercegovac and Tomás Lang. *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, Boston, 1994. (cit. on p. 76).
- [GVK22] Rémi Garcia, Anastasia Volkova, and Martin Kumm. “Truncated Multiple Constant Multiplication with Minimal Number of Full Adders”. In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2022, pp. 263–267. (cit. on p. 75).

- [HV14] Javier Hormigo and Julio Villalba. "Optimizing DSP Circuits by a New Family of Arithmetic Operators". In: *Asilomar Conference on Signals, Circuits and Systems*. IEEE, 2014, pp. 871–875. (cit. on pp. 71, 73).
- [HV16a] Javier Hormigo and Julio Villalba. "Measuring Improvement When Using HUB Formats to Implement Floating-Point Systems under Round-to-Nearest". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.6 (2016), pp. 2369–2377. (cit. on pp. 71, 73).
- [HV16b] Javier Hormigo and Julio Villalba. "New formats for computing with real numbers under round-to-nearest". In: *Transactions on Computers* 65.7 (2016), pp. 2158–2168. (cit. on pp. 71, 73).
- [Kor02] Israel Koren. *Computer Arithmetic Algorithms*. 2nd ed. Prentice-Hall, 2002. (cit. on p. 76).
- [MH15] S.D. Muñoz and Javier Hormigo. "Improving fixed-point implementation of QR decomposition by rounding-to-nearest". In: *International Symposium on Consumer Electronics (ISCE)*. 2015. (cit. on pp. 71, 73).
- [MKC09] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to interval analysis*. SIAM, 2009. (cit. on p. 71).
- [Mul+18] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. 2nd ed. Birkhäuser Boston, 2018. (cit. on p. 69).
- [Mul05] Jean-Michel Muller. *On the definition of ulp(x)*. Tech. rep. 2005-09. LIP Laboratory, ENS Lyon, 2005. URL: <http://www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2005/RR2005-09.pdf>. (cit. on p. 68).
- [Mul16] Jean-Michel Muller. *Elementary Functions, Algorithms and Implementation*. 3rd ed. Birkhäuser Boston, 2016. (cit. on p. 76).
- [Par10] Behrooz Parhami. *Computer Arithmetic, Algorithms and Hardware Designs*. 2nd ed. Oxford University Press, 2010. (cit. on p. 76).
- [SS99] Michael J. Schulte and James E. Stine. "Approximating Elementary Functions with Symmetric Bipartite Tables". In: *IEEE Transactions on Computers* 48.8 (1999), pp. 842–847. (cit. on p. 71).



4

CHAPTER 4

Field Programmable Gate Arrays

Field programmable gate arrays (FPGAs) are a natural target for application-specific arithmetic, and many of the techniques developed in this book were motivated by FPGA applications. This chapter gives a brief overview of FPGA architectural features relevant to application-specific arithmetic.

The first commercial FPGA in its current form was the XC2000 series introduced by Xilinx Inc. in 1985 [Bro+14]. What differentiates an FPGA from earlier programmable logic is its structure. As shown in Fig. 4.1, an FPGA is an array of basic logic elements (BLEs) embedded in a programmable interconnect network. A BLE can compute simple logic functions and typically also includes some flip-flops (FFs) for storing intermediate results.

Both the logic function computed by each BLE and the way they are interconnected can be configured after manufacturing the chip. This configuration is held in static RAM (SRAM) cells; it can be loaded in a few milliseconds. After this, the FPGA will behave like the logic circuit it was configured to emulate. Thanks to this flexibility, FPGAs have become a popular alternative to application-specific integrated circuits (ASICs). Programmability has a huge overhead over ASIC in terms of silicon area and performance, but this is partly compensated by very high-volume productions, which gives access to the smallest and fastest technologies at low unit cost. Current FPGAs include millions of BLEs and can work at frequencies close to 1 GHz.

More details about FPGA fundamentals can be found in excellent textbooks [Bro+14; BRM99]. This chapter focuses on describing FPGA features that are relevant to application-specific arithmetic. A first reason is that circuits configured on FPGAs are, very often, application-specific. Since the FPGA can be reconfigured when the application evolves, each configuration may only include those operators needed for the current context and fine-tuned to this context. A second reason is that FPGAs also have their

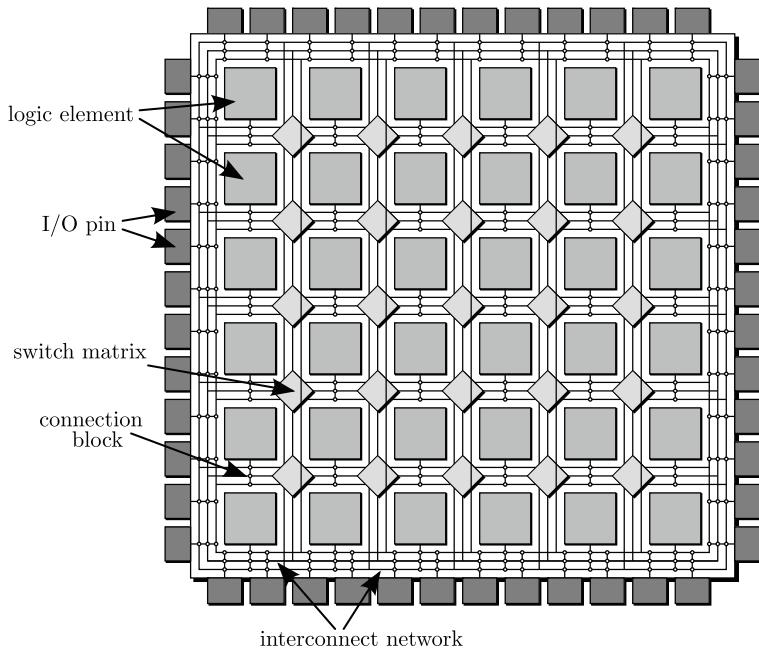


Fig. 4.1 Simplified architecture of a generic FPGA layout.

specificities. The best logic circuit realizing a given function in an ASIC may perform poorly in an FPGA and vice versa. The purpose of this chapter is to describe finely the FPGA features relevant to arithmetic, so that operators may be optimized for FPGAs.

Section 4.1 describes in more details the structure of BLEs in modern FPGAs. Section 4.2 describes the multiplier resources that were later added to the logic fabric. They are usually called DSP blocks because they were initially designed for digital signal processing (DSP). Section 4.3 describes the various memorization elements in FPGAs. In this book, they are very useful as read-only memories (ROMs) to store pre-computed values.

Two big companies currently lead the FPGA market: AMD (formerly Xilinx) and Intel (formerly Altera). This chapter provides some details on recent devices from these two manufacturers. We are aware that the probability is very high that device-specific data presented here will be outdated very soon after printing this book, and therefore we focus on features that have been stable for some time. Other companies, e. g., Menta, Microsemi, Lattice, Microchip, and QuickLogic, build FPGAs with similar structures. We hope it will be easy for the reader to transpose the knowledge of this chapter to these FPGAs.

The routing network is important, since it represents the bulk of the area and complexity of an FPGA chip and also an increasing proportion of the de-

lay of the circuits implemented in these chips. However, we will not attempt to detail its organization, firstly because routing resources are scarcely documented and secondly because they are well hidden behind vendor backend tools. Similarly, we do not expand on the very complex and very flexible input/output features of actual FPGA circuits. Although these are well documented, they are of little relevance to application-specific arithmetic.

4.1 Basic Logic Elements

We first describe the generic features of FPGA logic resources, before detailing vendor-specific details in Sects. 4.1.2 and 4.1.3.

4.1.1 Look-Up Tables and Flip-Flops

The minimal BLE consists of a look-up table (LUT) and an optional flip-flop (FF) as shown in Fig. 4.2.

The FF will be used if the SRAM cell (shown as an empty white block) contains a logic 1; otherwise, the BLE is purely combinatorial.

A LUT with α inputs is denoted by $\text{LUT}\alpha$ in the following. At runtime, it behaves as a small ROM addressed by the α inputs (4 in Fig. 4.2) that holds 2^α bits. This ROM may hold any arbitrary truth table; therefore the LUT can realize any Boolean function with α inputs. For instance, the BLE in Fig. 4.2 can realize any 4-input Boolean function. Functions with more than α inputs have to be mapped to several LUTs connected via the routing network.

The implementation of a LUT4 is illustrated in Fig. 4.3. It consists of 16 SRAM cells storing the $2^4 = 16$ possible outputs of the function. The inputs $a \dots d$ select the cell that is output using a multiplexer (MUX). From this

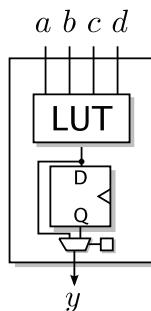


Fig. 4.2 Minimal BLE consisting of a look-up table with optional flip-flop.

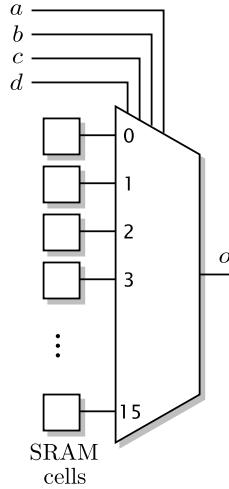


Fig. 4.3 Realization of a 4-input look-up table.

figure, it is clear that the input size α is a crucial parameter as the number of SRAM bits and the MUX size grow with 2^α . Architectural explorations on real benchmark circuits retrieved that the best area-delay products are achieved for input sizes between 4 and 6 inputs [AR04]. These are also the typical LUT sizes found in modern commercial FPGAs.

A LUT_α can be realized by two LUTs with $\alpha - 1$ inputs using Boole's expansion theorem. For $\alpha = 4$, this results in

$$f(a, b, c, d) = \bar{d} \wedge f_{\bar{d}}(a, b, c) \vee d \wedge f_d(a, b, c) \quad (4.1)$$

where $f_{\bar{d}}$ and f_d are obtained from evaluating $f(a, b, c, d)$ for $d = 0$ and 1, respectively,

$$f_{\bar{d}}(a, b, c) = f(a, b, c, 0) \quad (4.2)$$

$$f_d(a, b, c) = f(a, b, c, 1). \quad (4.3)$$

Hence, an additional MUX can be used to select between $f_{\bar{d}}$ and f_d . An example for the LUT4 built from two LUT3 and a MUX is shown in Fig. 4.4.

This trick is often used in commercial FPGAs to get more flexibility in their BLEs by trading between either many LUTs with fewer (potentially shared) inputs or fewer LUTs with many inputs.

In modern commercial FPGAs, the BLE is always enhanced with special support for the addition operation. The main idea here is to have a dedicated link to propagate a carry bit from one BLE to one of its direct neighbors, bypassing the generic programmable routing and the associated delay. This

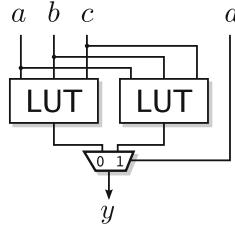


Fig. 4.4 Realization of a LUT4 by using two LUT3 using Boole's expansion theorem.

feature is commonly called *fast carry chain* or *fast carry logic*. Its implementation varies in FPGAs from different vendors, as the next sections show.

4.1.2 Basic Logic Elements of AMD FPGAs

On AMD FPGAs, the BLEs are grouped in logic clusters called Configurable Logic Blocks (CLBs), and a CLB is itself divided into sub-units called slices.

We first focus on the old (but simple) Virtex 4 CLB, which contains four slices. Each Virtex 4 slice contains two BLEs consisting of a LUT4, a carry-chain logic and a single flip-flop (FF) [Xil08]. The simplified structure of a Virtex 4 BLE is shown in Fig. 4.5a (SRAM cells that control the MUXes are omitted from now on to simplify the figures). It can realize any Boolean function of four input variables ($a \dots d$), followed by an optional register (selected by the MUX close to the output y). The BLE also contains a dedicated carry logic consisting of an exclusive-or (XOR) gate and a MUX. How this is used to build adders is discussed in Sect. 5.4.1. The optional AND gate of the Virtex 4 BLE can be used for generating partial products as needed in generic multiplications.

Starting from the Virtex 5 architecture, the LUT was extended to a LUT6 per BLE which can be configured into two independent 5-input LUTs that share the same inputs as shown in Fig. 4.5b [Xil07]. In the 6 and 7 series as well as the Ultrascale(+) FPGA architectures, an additional FF was added (shown gray in Fig. 4.5b) [Xil12; Xil10; Xil12a]. They also contain the same carry logic as the Virtex 4, but the AND gate is omitted, probably because there are better ways to exploit LUTs for multiplications (see Chap. 8). Each slice of the series 5/6/7 architecture contains four identical BLEs. In the UltraScale/UltraScale+ generations, the BLE is essentially similar, but a slice now contains eight BLEs instead of four, and one CLB contains only one slice, plus routing resources.

Further extensions have been made with the Versal architecture [AMD23a; AMD22b] in which simplified BLE is shown in Fig. 4.5c. The main change is that both 5-input LUTs are further separated into two 4-input LUTs. This

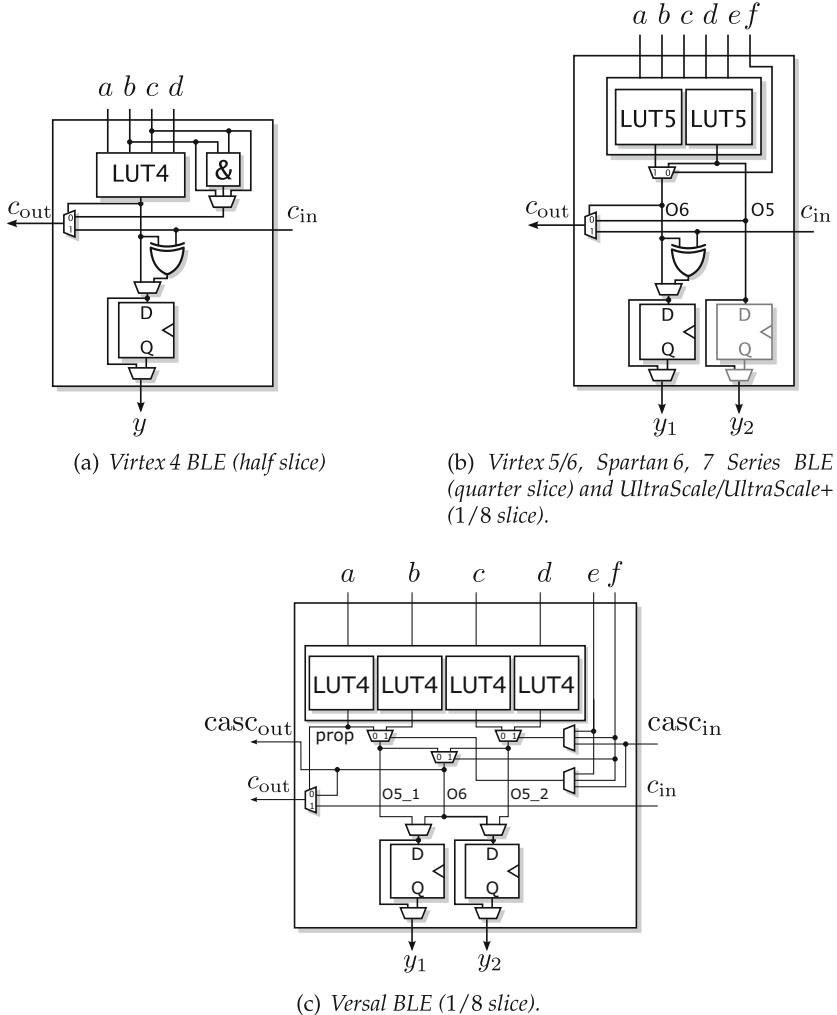


Fig. 4.5 The evolution of BLEs in AMD FPGA families.

allows the computation of the XOR gate within one of the 5-input LUTs to build adders as we will detail in Sect. 5.4.1. To do so, the output of the LUT6 (O6) can be routed into the neighboring BLE using a special cascade connection (casc_out and casc_in). The main simplification in Fig. 4.5c comes from the fact that the carries are actually computed from a fast carry-lookahead logic (see Sect. 5.3.4). However, the shown single MUX is logically equivalent.

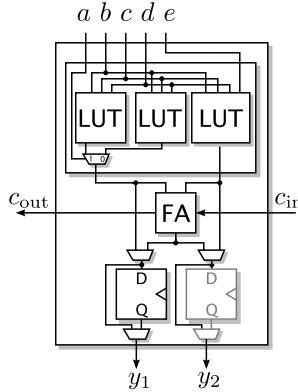


Fig. 4.6 Simplified BLE (half ALM) of Intel Stratix III to Stratix V and Stratix 10.

4.1.3 Basic Logic Elements of Intel FPGAs

The logic cluster in Intel Stratix FPGAs is called logic array block (LAB), and its sub-units are called adaptative logic modules (ALMs). The Stratix III to Stratix V as well as the Stratix 10 FPGAs contain ten adaptative logic modules (ALMs) per LAB [Alt13]. The functionality of a BLE as defined above is roughly covered by one half of an ALM. This Intel BLE mainly consists of one LUT4, two 3-input LUTs, a dedicated full adder (FA), and two FFs as shown in Fig. 4.6. The second FF (shown gray in Fig. 4.6) was introduced with the Stratix V architecture and is missing in older Stratix FPGAs. Each ALM has eight input lines which have to be shared between two BLEs. Multiplexers between both BLEs in the same ALM can be used to adapt the logic granularity to different effective LUT sizes. The possible configurations of a complete ALM (two BLEs) are:

- two independent 4-input LUTs,
- two independent 3-input and 5-input LUTs,
- one independent 6-input LUT,
- various combinations of LUTs up to six inputs where some of the inputs are shared,
- and some specific 7-input LUTs [Alt13].

With the Stratix 10 (which is the successor of Stratix V), the ALM was simplified compared to previous generations. The two 3-input LUTs are replaced by a single 4-input LUT, with slightly fewer LUT configurations than before [Int17].

4.1.4 Comparison of BLEs in Commercial FPGAs

Comparing the BLEs of different generations of different manufacturers in Figs. 4.5 and 4.6, one can conclude that all BLE follow a common pattern: They consist of:

1. a first layer of LUT(s) that can be subdivided into smaller LUTs;
2. a fast carry chain to speed up addition;
3. and finally, optional FF(s) to store the results.

Hence, one can expect that future FPGAs will look similar. Throughout this book, we will provide many examples for specific commercial FPGAs with the hope that the ideas can be adapted to future FPGAs as well.

4.2 DSP Blocks

A DSP block typically contains a multiplier where one input can come from an optional pre-addition, and the output can be followed by an optional post-addition, performing the operation

$$R = (X_0 + X_1) \times Y + Z. \quad (4.4)$$

The post-adder can be used to combine results from several DSP blocks to build larger multipliers. Another use of the post-adders are generic multiply-accumulate (MAC) operations as required in operations such as complex multiplication or polynomial evaluation, but also in applications like digital filters.

The main motivation for embedded pre-adders in the DSP block is their use in digital filters (for symmetric finite impulse response (FIR) filters). They can be also used to build efficient large multipliers using the Karatsuba algorithm that will be presented in Sect. 8.6. Another use is the complex product $P_r + jP_i$ of two complex inputs $A_r + jA_i$ and $B_r + jB_i$ in 3 scalar multiplications and 5 scalar additions, using the relations

$$P_r = A_r B_r - A_i B_i \quad (4.5)$$

$$= \textcolor{blue}{A_r(B_r + B_i)} - (A_r + A_i) B_i \quad (4.6)$$

$$P_i = A_r B_i + A_i B_r \quad (4.7)$$

$$= \textcolor{blue}{A_r(B_r + B_i)} - (A_r - A_i) B_r \quad (4.8)$$

where the term in blue is shared between P_r and P_i and computed only once.

The main difference between the DSP blocks of different FPGAs are the supported word sizes. AMD FPGAs typically provide fixed-size units which can be combined to build larger multipliers. Intel FPGAs are more flexible,

Table 4.1 Supported multiplier configurations in current commercial FPGAs of Xilinx/AMD (top) and Intel (bottom).

Device	Units per DSP	Multiplier	pre-add	post-add
Virtex 5	1	18×25	–	48 bit
Virtex 6,7	1	18×25	25 bit	48 bit
Ultrascale(+)	1	18×27	27 bit	48 bit
Stratix V	3	9×9	–	–
Stratix V	2	16×16	18 bit	– ^a
Stratix V	1	27×27	27 bit	64 bit
Stratix V	1	36×18	–	64 bit
Stratix 10	2	18×18	18 bit	– ^a
Stratix 10	1	27×27	26 bit	64 bit

^aThere is a single post-adder that can be used to sum the two multiplier results

supporting different sizes and more combinations of multipliers within a DSP block.

Table 4.1 provides an overview of the sizes of multipliers, pre-adders, and post-adders found in more recent FPGAs. Note that Intel FPGAs typically support signed and unsigned multiplications, while the size for the AMD FPGAs is given for signed only. Since a signed multiplier can be used as an unsigned multiplier by setting the most significant bit (MSB) to zero (the sign bit), the resulting word sizes of the operands will be one bit less for unsigned multiplication.

Intel FPGAs allow two multiplications per DSP with a smaller word size (16×16 for Stratix V and 18×19 for Stratix 10). In this mode, the post-adder can be used to sum the results of the two independent multipliers. This is in particular useful for complex multiplications using (4.5) and (4.7): only two DSPs are required to compute a complex multiplication, one for the real part and one for the imaginary part.

The DSP blocks of the Stratix 10 FPGAs are very specific as they also additionally support hardened floating-point operations in single precision. Here, each DSP block can be configured to perform a single precision multiplication, or addition/subtraction, or MAC operation [LP15].

4.3 Memory Elements

FPGAs provide dedicated structures for realizing addressable memory. *Distributed memory* uses the configuration memory of BLEs to build small memories. For larger memories, *memory blocks* are dedicated memory units, similar to the DSP blocks, and similarly configurable. These memories are essentially used in applications to store runtime data. However, they can also contain tables of precomputed values, an important ingredient of application-

specific arithmetic. This section therefore reviews the main properties of these memory resources.

4.3.1 Distributed Memory

The slices in AMD FPGAs can be of two different types: logic slices (SLICEL) and memory slices (SLICEM). The SLICEL contains the BLEs as described in Sect. 4.1.2. The SLICEM is a superset of the SLICEL that additionally supports the realization of distributed memory. Although not documented at that level of detail, a simple model can be derived for a LUT5 of a SLICEM. It is shown in Fig. 4.7a. It consists of the same structure as the generic LUT shown in Fig. 4.3, but the SRAM cells are here FFs that can be written at run-time using an additional data input DI1. The selection of the FF being written to is performed by write address (WA) which enables the corresponding FF using a one-hot-decoder. The common LUT inputs are used as a read address. As those read and write ports are independent, this memory is called a *simple* dual-port memory. Each LUT6 of a SLICEM contains two LUT5 and can thus realize two 32×1 bit memories. Alternatively, a single 64×1 bit memory can be realized per LUT6. By combination of these primitive memory cells, several memory configurations can be constructed.

Similar to the AMD FPGAs, Intel FPGAs include memory LABs (MLABs) to support distributed random access memory (RAM). MLABs are a superset of the LABs as described in Sect. 4.1.3. Each ALM of the MLABs can realize a 32×2 bit simple dual-port memory block. Like for AMD, several ALMs in the MLABs can be combined to build larger memories.

4.3.2 Shift Register LUTs

Another configuration mode of the SLICEM discussed above is the shift register LUT (SRL). The LUT5 model of this configuration is shown in Fig. 4.7b. Here, the FFs build a shift register by chaining all the FFs. The input of the shift register is the DI1 input or, in case of cascading several LUT5s in SRL mode, the MC31 output of the cell below. Now, the LUT inputs can select any of the delayed versions of DI1 during run time (having at least a delay of one). This feature is often used when a signal has to be delayed by several cycles, e. g., for pipeline synchronization.

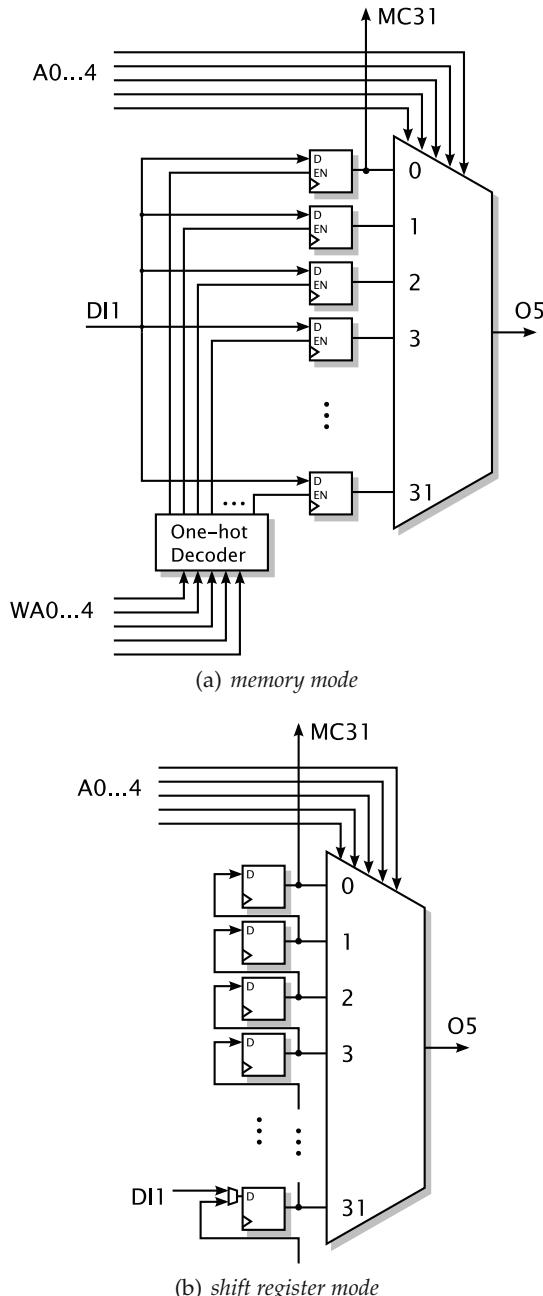


Fig. 4.7 LUT5 found in a SLICEM slice in modern AMD FPGAs.

Another feature of this mode is that the FF contents can be programmed sequentially. This allows to re-program the LUT during runtime within 32 clock cycles using a single wire. It is one way to perform dynamic runtime reconfiguration in which the overall structure stays the same but LUT contents are changed. This may especially happen in application-specific arithmetic where, e.g., a coefficient has to be replaced from time to time by another [Mö17; Har+19].

4.3.3 Block Memory

While distributed memory provides pervasive cheap small memories, it is costly to build larger memories from them. To support larger memories, FPGA manufacturers also embed block RAMs in the FPGA fabric, in the same way they embed DSP blocks.

For Xilinx/AMD 7th generation and Ultrascale FPGAs, a block RAM has a capacity of 36 kbits. The block RAMs of Intel FPGAs (Stratix 5 and 10th generation of FPGAs) have a capacity of 20 kbits. In both cases, the block RAM can be configured to support different trade-offs between memory depth and data word size. The supported configurations are given in Table 4.2.

Block RAMs are usually true dual-port memories, supporting read and write on both ports, but some configurations only allow the simple dual-port interface with one read port and one write port.

Large FPGAs provide thousands of block RAM cells, providing a total memory capacity of several Mbytes.

Concerning performance, block RAMs can achieve the nominal frequency of the FPGA (which more or less corresponds to the delay of one logic element). However, each memory access then requires several cycles, using dedicated internal pipelining registers.

Some of the Stratix 10 also introduced the embedded SRAM (eSRAM), a large memory of about 5.90 Mb. They only support single port access, and there are only a few eSRAMs on the largest FPGAs. This resource is probably irrelevant to application-specific arithmetic.

Table 4.2 Supported block RAM configurations in commercial FPGAs of AMD/Xilinx (top) and Intel (bottom).

Device	RAM configuration [bit]
AMD/Xilinx 7th Gen. & Ultrascale	$32k \times 1$
	$16k \times 2$
	$8k \times 4$
	$4k \times 9$
	$2k \times 18$
	$1k \times 36$
Stratix 10 (M20K)	512×72
	$2k \times 10$
	$1k \times 20$
	512×40

References

- [Alt13] *Stratix V Device Handbook*. Altera Corporation. 2013. (cit. on p. 93).
- [AMD22b] *Versal Architecture Prime Series Libraries Guide (UG1344)*. Advanced Micro Devices, Inc. 2022. (cit. on p. 91).
- [AMD23a] *Versal ACAP Configurable Logic Block Architecture Manual (AM005)*. Advanced Micro Devices, Inc. 2023. (cit. on p. 91).
- [AR04] Elias Ahmed and Jonathan Rose. “The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12.3 (2004), pp. 288–298. (cit. on p. 90).
- [BRM99] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999. (cit. on p. 87).
- [Bro+14] Stephen D Brown, Robert J Francis, Jonathan Rose, and Zvonko G Vranesic. *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, 2014. (cit. on p. 87).
- [Har+19] Martin Hardieck, M Kumm, Konrad Möller, and Peter Zipf. “Reconfigurable Convolutional Kernels for Neural Networks on FPGAs”. In: *International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM, 2019, pp. 43–52. (cit. on p. 98).
- [Int17] *Intel Stratix 10 Logic Array Blocks and Adaptive Logic Modules User Guide*. Intel Corporation. 2017. (cit. on p. 93).
- [LP15] Martin Langhammer and Bogdan Pasca. “Floating-Point DSP Block Architecture for FPGAs”. In: *International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM Press, 2015, pp. 117–125. (cit. on p. 95).

- [Möl17] Konrad Möller. “Run-time Reconfigurable Constant Multiplication on Field Programmable Gate Arrays”. PhD thesis. Kassel University Press, 2017. (cit. on p. [98](#)).
- [Xil07] *Virtex-5 FPGA User Guide (UG190)*. Xilinx. 2007. (cit. on p. [91](#)).
- [Xil08] *Virtex-4 FPGA User Guide (UG070)*. Xilinx. 2008. (cit. on p. [91](#)).
- [Xil10] *Spartan-6 FPGA Configurable Logic Block User Guide (UG384)*. Xilinx. 2010. (cit. on p. [91](#)).
- [Xil12] *Virtex-6 FPGA Configurable Logic Block User Guide (UG364)*. Xilinx Inc. 2012. (cit. on p. [91](#)).
- [Xil12a] *7 Series FPGAs Configurable Logic Block Users Guide (UG474)*. Xilinx. 2012. (cit. on p. [91](#)).

Part I

Revisiting Classic Arithmetic



CHAPTER 5

Fixed-Point Addition

*They were rescued three days later, emaciated but beaming with joy:
they had just invented a new method, deduced from the logarithmic spiral,
for adding one-digit integers.*

Topfer

Integer addition is without doubt the most versatile arithmetic operation. Integer and fixed-point adders will be used as building blocks in most chapters of this book. The present chapter reviews the main techniques for adder construction, from simple but slow adders to larger and faster ones. Integer addition also received special attention in most FPGA architectures, and this chapter reviews in depth the specificities of addition in FPGAs.

Application-specific arithmetic requires efficient adders of all sizes, from a few bits to a few hundred bits. Small adders are required for low-precision signal processing, but also, for instance, in counters used for datapath control and in the exponent processing of floating-point computations – exponents sizes typically vary between 5 bits, for a dynamic range of $[10^{-6}, 10^6]$, and 16 bits, for a dynamic range beyond $[10^{-10,000}, 10^{10,000}]$.

Larger adders (32 to 128 bits) are used everywhere. A modern microprocessor routinely performs several integer 32-bit or 64-bit additions per cycle: for address computation (on the program counter, the stack counter, the various addressing modes, etc); to process loop counters; and of course to process integer or fixed-point data. Besides, integer adders are used to build more complex operators, in particular multipliers and dividers of integer or floating-point data. Most techniques for the evaluation of numerical functions (reviewed in Part III of this book) also involve additions of various sizes. In short, integer or fixed-point addition is used as a building block in virtually all the coarser operators.

Very large (more than 128 bits) adders are used in the construction of modular adders for cryptography. In cryptography based on the Rivest-Shamir-Adleman (RSA) algorithm, the acceptable key sizes are beyond 1024 bits. Elliptic-curve cryptography works with smaller data sizes, but even the acceptable security requires more than 150 bits [Oka+00].

Integer addition is so pervasive that it is generally supported by hardware description languages. One may just write a `+` in VHDL or Verilog, and synthesis tools will instantiate an adder.

In many cases (small additions, and in general additions that are not in the critical path), one does not need to worry about the implementation of such an addition. The most area-efficient adder implementation (a ripple-carry adder; see Sect. 5.2.2) will be adequate, and synthesis tools will infer just that. On FPGAs, the ripple-carry adder benefits from dedicated carry logic that makes it very efficient even in terms of speed.

Conversely, when addition speed is critical, it is possible to design faster (but more expensive) adders. The approaches used here are quite different when targeting ASIC or FPGA.

ASIC synthesis tools may use various techniques that reduce the addition latency at the expense of area. These techniques include transistor-level improvements to the carry-propagation path, but also a set of algorithmic techniques referred to in the literature as *fast adders*. These techniques will be reviewed in Sect. 5.3.

On FPGAs, classic fast adder techniques are mostly ineffective, due to the performance discrepancy between fast carry routing and general routing. The frequency of large ripple-carry adders may be improved by pipelining, which will be reviewed in Sect. 5.4. FPGA-specific fast addition techniques have also been studied and will be surveyed in Sect. 5.5.

5.1 Fixed-Point Considerations

In Chap. 2, integers have been presented as a special case of fixed-point representation. In this section, we first discuss integer addition and then generalize it to fixed-point addition.

5.1.1 Unsigned Integer Addition

An unsigned integer on w bits belongs to the interval $[0, 2^w - 0]$. The sum of two such integers therefore belongs to $[0, 2^{w+1} - 2]$. It fits a $w + 1$ -bit integer, and there is even room for adding one more unit to fill the interval $[0, 2^{w+1} - 0]$. This defines the generic binary integer adder depicted in Fig. 5.1a: it inputs two w -bit unsigned integers $X = \sum_{i=0}^{w-1} 2^i x_i$ and $Y =$

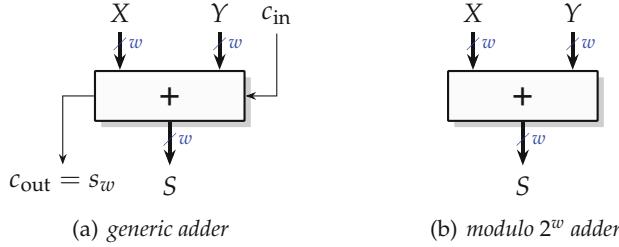


Fig. 5.1 Functional views of the integer adder.

$\sum_{i=0}^{w-1} 2^i y_i$, and an input carry bit c_{in} of weight 1. It computes $X + Y + c_{in}$ exactly, as a $(w+1)$ -bit integer $S = \sum_{i=0}^w 2^i s_i$. The most significant output bit s_w is usually called the *output carry* and denoted as c_{out} . It is the case in Fig. 5.1a.

When this carry-out bit is ignored, the operation computed by the adder is $X + Y + c_{in} \bmod 2^w$. Figure 5.1b shows an adder with only two inputs and one output, all of the same size. It is a special case of Fig. 5.1a where $c_{in} = 0$ and c_{out} is dropped, and it computes $X + Y \bmod 2^w$.

In some applications, in particular in signal processing, the interface seen in Fig. 5.1b is imposed, but *saturation* is preferred to this modulo behavior. A saturated adder returns the maximal representable value in case of overflow. It can be built on top of a generic adder: a multiplexer, controlled by c_{out} , outputs the w -bit sum if $c_{out} = 0$ and outputs 2^{w-1} if $c_{out} = 1$.

5.1.2 Signed Integer Addition

A signed integer on w bits belongs to the interval $[-2^{w-1}, 2^{w-1} - 0]$. The sum of two such integers therefore belongs to $[-2^w, 2^w - 2]$. Again it fits a $w+1$ -bit signed integer, with the possibility to add a unit carry-in. The management of signs in integer addition will be detailed in Sect. 5.2.3, which will also show how subtraction also reduces to addition in two's complement.

5.1.3 Fixed-Point Addition

An unsigned fixed-point numbers in the format $\text{ufix}(m, \ell)$ is an integer of size $w = m - \ell + 1$ scaled by 2^ℓ (see Sect. 2.2.1, p. 38). Therefore, the addition of two unsigned fixed-point numbers X and Y of the same format $\text{ufix}(m, \ell)$ can be reduced to an integer addition:

$$S = 2^\ell X_{\text{int}} + 2^\ell Y_{\text{int}} = 2^\ell(X_{\text{int}} + Y_{\text{int}}) \quad (5.1)$$

Since $X_{\text{int}} + Y_{\text{int}}$ is a $w + 1$ -bit integer, the exact fixed-point sum is an unsigned number in $\text{ufix}(m + 1, \ell)$: it has one bit more at the most significant bit (MSB) than the inputs.

The addition of two numbers of different formats $X \in \text{ufix}(m_X, \ell_X)$ and $Y \in \text{ufix}(m_Y, \ell_Y)$ is best understood by first converting them to a larger format which allows for an exact addition (see Fig. 5.2). This requires sign extension if the numbers are signed (see Sect. 2.2.3, p. 42). Defining $m = \max(m_X, m_Y) + 1$ and $\ell = \min(\ell_X, \ell_Y)$, then the exact sum can be represented in a $\text{ufix}(m, \ell)$ format.

If the sum S is expected in a smaller format $\text{ufix}(m_S, \ell_S)$, then overflow may happen if $m_S < m$ and rounding must happen if $\ell_S > \ell$.

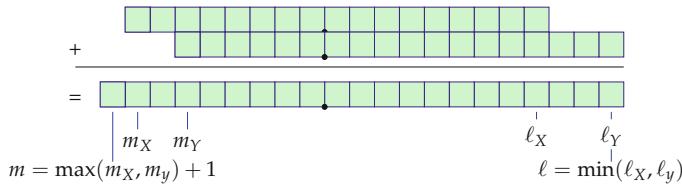


Fig. 5.2 Exact addition of two unsigned fix-point numbers of different formats.

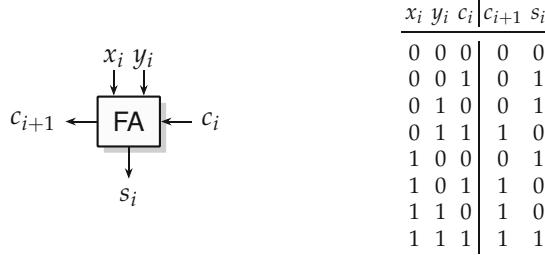
Since fixed-point addition can in all cases be reduced to integer addition, the remainder of this chapter focuses on integer addition.

5.2 Addition Basics

5.2.1 The Full Adder and Half Adder Cells

The full adder (FA) performs the analogous to a one-digit step of the paper-and-pencil addition algorithm. It computes the sum of its three input bits x_i , y_i , and c_i , where i denotes the bit position corresponding to weight 2^i . This sum is between 0 and 3 and may therefore be rewritten as a 2-bit binary number $c_{i+1}s_i$ where c_{i+1} is the most significant bit and s_i the least significant bit. Figure 5.3 describes the black box of the full adder (FA) and its truth table. In general c_i will be referred to as the input carry, and c_{i+1} is the output carry.

Looking at the truth table, the sum bit is the exclusive OR of the three inputs. The carry bit is the majority function, which is true when more than half of the inputs are true. This logic function can also be defined as the basic Boolean expressions

**Fig. 5.3** The full adder.

$$s_i = x_i \oplus y_i \oplus c_i \quad (5.2)$$

$$c_{i+1} = x_i c_i \vee y_i c_i \vee x_i y_i. \quad (5.3)$$

We will sometimes denote the full adder according to (5.2) and (5.3) as

$$(c_{i+1}, s_i) = \text{FA}(x_i, y_i, c_i). \quad (5.4)$$

As the FA performs the sum of its three inputs, these three inputs are functionally equivalent. However, in general, one wishes to reduce the delay of a complete w -bit addition, which happens when the carry is propagated. In the full adder cell, this means accelerating the path from c_i to c_{i+1} . Therefore, in terms of implementation, the three inputs are usually not equivalent.

There are many Boolean expressions equivalent to (5.3) and even more transistor-level implementations of these expressions. Actually, the FA cell is so important that founders provide several full-custom implementations for their technologies. Several papers and patents are published every year about the design of FA cells [ZW92; AB95; AB95; SB00; BWJ03; Bha+15].

In case one input is zero, the full adder reduces to a half adder (HA). Figure 5.4 shows the HA, and its truth table, which is identical to the FA for $c_i = 0$. Its sum becomes an XOR and its carry-out simplifies to an AND:

$$s_i = x_i \oplus y_i \quad (5.5)$$

$$c_{i+1} = x_i y_i. \quad (5.6)$$

**Fig. 5.4** The half adder.

5.2.2 Ripple-Carry Addition

The most area-efficient (but also slowest) adder is the ripple-carry adder (RCA) shown in Fig. 5.5.

The critical path of this design goes from one of the inputs of the rightmost FA (corresponding to the least significant bit) to one of the outputs of the leftmost one (most significant bit). In general, the latency of such an adder is linear in the number of bits w

$$\tau_{\text{add}}(w) = (w - 1)\tau_{\text{cp}} + \max(\tau_{\text{cp}}, \tau_{\text{fa}}) \quad (5.7)$$

where τ_{fa} is the delay from inputs to outputs of a full adder cell and τ_{cp} is the carry propagate delay (from c_i to c_{i+1}). The RCA belongs to the class of carry propagate adders (CPAs). Other, faster types of CPAs are discussed in Sect. 5.3.

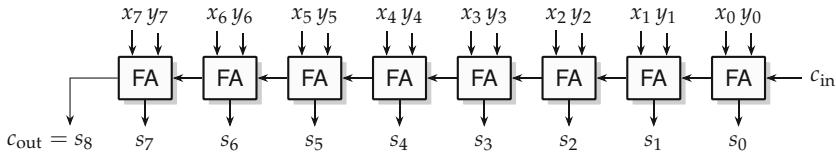


Fig. 5.5 An 8-bit ripple-carry adder.

5.2.3 Addition of Signed Numbers in Two's Complement

If the carry-in bit is set to 0 and the carry-out bit is ignored, the adder of Fig. 5.5 inputs two w -bit vectors X and Y and outputs a w -bit vector S . When these three vectors X , Y , and S are interpreted as unsigned w -bit integers (in $\text{ufix}(w - 1, 0)$ format), then this operator computes in S the sum modulo 2^w of X and Y .

This property also holds if X , Y , and S are interpreted as *signed* w -bit integers (in $\text{sfix}(w - 1, 0)$ format). Indeed, two's complement arithmetic on w bits is really modulo 2^w arithmetic (see Fig. 2.2), with the choice that the numbers with their MSB set represent negative numbers instead of large positive ones.

This is actually the main advantage of two's complement: the adder of Fig. 5.5 can be used unmodified to add signed numbers.

The only difference lies in the interpretation of the carry-out bit:

- For unsigned arithmetic, a carry-out of $c_{\text{out}} = 1$ indicates an overflow.
- For signed arithmetic, there will be an overflow if the two inputs have the same sign bit and the output has a different sign bit. Otherwise, if the two

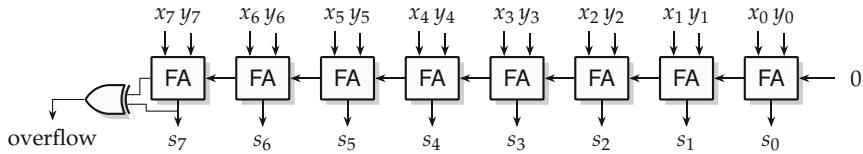


Fig. 5.6 An 8-bit adder for two's complement numbers. The FA inputting a zero carry on the right can be replaced by a less expensive HA.

inputs have different sign bits, then their sum cannot overflow. A classic result is that an overflow bit can be computed as the XOR of s_{w-1} (the sign of the result) and $s_w = c_{\text{out}}$, as illustrated by Fig. 5.6. The proof is by enumeration of the various cases.

5.2.4 Basic Subtraction

Similar to the FA, one can derive a primitive circuit for performing subtractions which is called a *full subtractor*. This cell computes the difference as well as a *borrow* bit that works similar to the carry out, but has a negative weight.

However, most subtractors are built using a different approach. The subtraction operation can also be seen as the addition of the negative subtrahend, i.e., $X - Y = X + (-Y)$.

In two's complement, $-Y$ is computed as $\bar{Y} + 1$ where \bar{Y} denotes the bitwise complement of Y . Technically, this is realized by inverting Y and adding a "1" to the least significant bit (LSB) position. This "1" can be added for free by using the carry-in of the adder, as illustrated by Fig. 5.7. Overflow detection logic does not appear on this figure, because it depends on the signedness of the inputs.

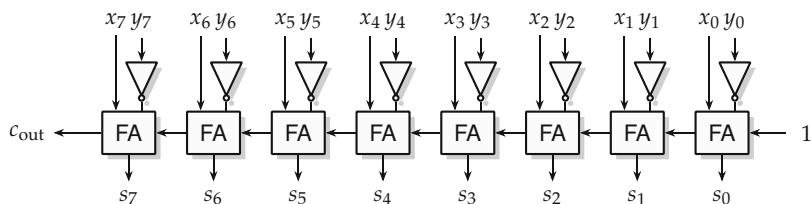


Fig. 5.7 An 8-bit subtractor computing a result in two's complement.

5.2.5 Reconfigurable Adder/Subtractor

The idea of the subtractor can be further generalized to an adder/subtractor. An adder/subtractor inputs an additional select signal sub and computes

$$S = \begin{cases} X + Y & \text{when } \text{sub} = 0 \\ X - Y & \text{when } \text{sub} = 1 \end{cases}. \quad (5.8)$$

In the case of subtraction, the two's complement of the Y input has to be computed: Y has to be bit-wise inverted, and a “1” has to be added at the LSB. The conditional inversion can be realized by a bit-wise XOR between y_i and sub . The conditional +1 can be realized by adding the “sub” signal using the carry-in as shown in Fig. 5.8.

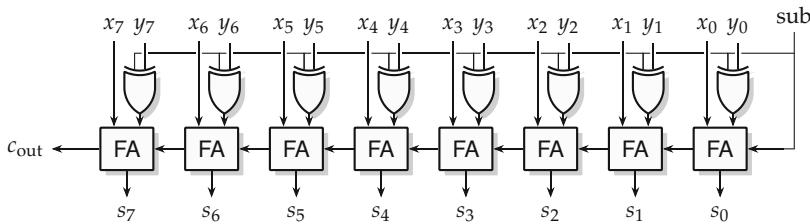


Fig. 5.8 Adder/subtractor computing $S = X + (-1)^{\text{sub}}Y$.

The main take-away message of Sects. 5.2.3, 5.2.4, and 5.2.5 is that thanks to two's complement, both addition and subtraction of signed or unsigned numbers reduce to the basic ripple-carry adder. They can therefore all benefit from the fast addition techniques discussed in the sequel.

5.2.6 Carry-Save Addition

The simplest variant of fast addition is actually a simplification of the ripple-carry adder: the *carry-save* adder depicted in Fig. 5.9 outputs all the carries instead of propagating them from one FA cell to the next. The unused carry-in signals of each FA cell together form a third binary input $Z = \sum_{i=0}^{w-1} 2^i z_i$. Thus, a w -bit carry-save adder inputs three w -bit numbers and outputs their sum as two w -bit numbers S and C .

In Fig. 5.9, the indices of all the bits correspond to their bit positions. There is no c_0 in Fig. 5.9, because the carry output of the rightmost FA cell already has weight 2. If we define $c_0 = 0$, then the bits c_i together define an integer $C = \sum_{i=0}^w 2^i c_i$. The relationship between the inputs and outputs of

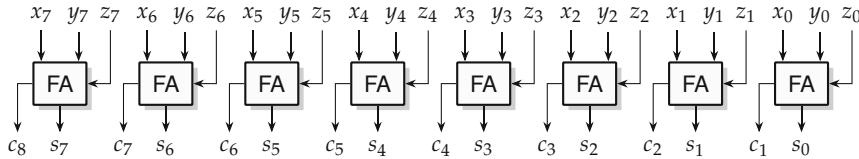


Fig. 5.9 An 8-bit carry-save adder.

the carry-save adder is then

$$C + S = X + Y + Z. \quad (5.9)$$

This addition is obviously not complete, since the output $C + S$ is still an unevaluated sum. However, the carry-save adder has definitely performed one addition, since there were two additions in $X + Y + Z$, and only one remains in the result $C + S$. Besides, this addition was performed with minimal hardware and with a delay independent of the addition size.

A number kept as an unevaluated sum $C + S$ is said to be in carry-save form. Carry-save is not a very interesting final format to express numbers, because it requires twice as many bits as standard binary to express the same range and because it is very redundant: there are many ways to write most integers in carry-save form,¹ which makes some operations such as comparisons very difficult. Actually the best way to compare two carry-save numbers is to first convert them to binary (by performing the addition $C + S$).

However, carry-save can be very useful for intermediate results, in particular in iterative algorithms: the use of carry-save form may allow for an iteration without carry propagation, thus dramatically reducing the iteration time. Classic examples include multipliers (see Chap. 8) and dividers (see Chap. 9).

A variant of carry-save consists in keeping α -bit carry propagation, outputting carries every α bits only, as illustrated in Fig. 5.10. This is called high-radix carry-save (HRCS), since the carries that are output are those of a radix- 2^α addition (see Sect. 2.3). For instance, for $\alpha = 4$, what we have is a hexadecimal carry-save adder. This is also called *partial redundant form* [FO01].

HRCS is less redundant than carry-save² by having fewer carry bits. This may be beneficial if these carries are to be stored in registers. However, the addition has a longer critical path (now that of a α -bit addition). Thus, it offers a trade-off between area and speed, which will be used, for instance, in Chap. 21.

¹ Referring to Sect. 2.4, carry-save is a binary system with the redundant digit set $\{0, 1, 2\}$ since each digit is represented by two bits.

² HRCS is a radix- 2^α system with the redundant digit set $\{0, 1, \dots, 2^\alpha\}$ since each digit is represented by one standard radix- 2^α digit, plus one unit bit.

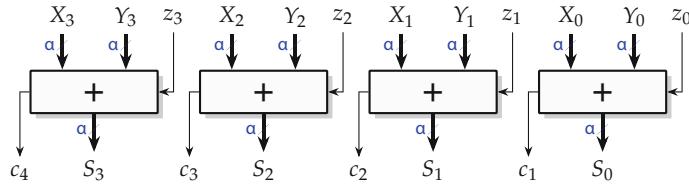


Fig. 5.10 A high-radix carry-save adder.

Carry-save arithmetic is not limited to the addition of three numbers: all these ideas will be generalized in Chap. 7.

5.3 Fast Adders for VLSI

The key observation behind the construction of fast adders is that the bulk of the delay comes from the carry propagation. Hence, a fast adder should accelerate the carry propagation. For that, it helps to study carry propagation as a function of the inputs x_i and y_i of the FA. There are three cases which are given in Table 5.1, a condensed version of the truth table of Fig. 5.3. In the first case, when $x_i = y_i = 0$, the output carry c_{i+1} will be zero, whatever the value of the input carry c_i . As this stops carry propagation, this is called the *kill* case. Similarly, in the last case when $x_i = y_i = 1$, a carry will be always generated whatever the value of c_i . The only cases of actual carry propagation are when exactly one of x_i or y_i is equal to 1. In this *propagate* case, the output carry is equal to the input carry.

Table 5.1 Cases in carry propagation in a full adder.

x_i	y_i	c_{i+1}	Case
0	0	0	kill
0	1	c_i	propagate
1	0	c_i	
1	1	1	generate

These cases can be encoded by the following Boolean expressions:

- kill: $k_i = \bar{x}_i \wedge \bar{y}_i = \bar{x}_i \veebar y_i$
- propagate: $p_i = x_i \oplus y_i$
- generate: $g_i = x_i y_i$

All these signals are independent from the carry propagation and can be obtained in parallel from the x_i and y_i only. Now, the carry and sum computations reduce to

$$c_{i+1} = g_i \vee p_i c_i \quad (5.10)$$

$$s_i = p_i \oplus c_i. \quad (5.11)$$

An alternative for the propagate case combines the propagate and generate cases as the *alive* case

$$a_i = \bar{k}_i = x_i \vee y_i \quad (5.12)$$

still enabling the definition of carry propagation as

$$c_{i+1} = g_i \vee a_i c_i. \quad (5.13)$$

An alternative expression for the generate can be obtained by considering that the generate case is only possible when not in the propagate case and one of the inputs is true (compare with Table 5.1):

$$g_i = \overline{p_i} x_i = \overline{p_i} y_i \quad (5.14)$$

This is useful for multiplexer (MUX)-based implementations of the carry computation, where p_i serves as the select input:

$$c_{i+1} = \overline{p_i} x_i \vee p_i c_i = \overline{p_i} y_i \vee p_i c_i \quad (5.15)$$

It is clear from (5.11) that each sum bit can be obtained from the corresponding carry and propagate bits. If we are able to compute all the carries, then we also have all the sums, at the cost of just another exclusive-or (XOR) delay. All the approaches discussed in the following are based on a fast computation of all the carries, where “fast” means “faster than in the ripple-carry adder.”

5.3.1 Switched Ripple-Carry Adder

The idea of this adder is to use fast switches to realize the three cases of carry propagation discussed above. The speedup here comes from the fact that pure switches may be much faster than complex logic gates. Its structure is identical to the RCA, but each full adder is realized using fast switches. The FA cell is shown in Fig. 5.11. First, the GKSSP block computes the generate, kill, and propagate signals. These mutually exclusive signals now control the fast switches to either pass the input carry to the output ($p_i = 1$) or to clear ($k_i = 1$) or set ($g_i = 1$) the output carry by pulling or pushing the signal to a logic high (supply voltage, Vcc) or low (ground, GND).

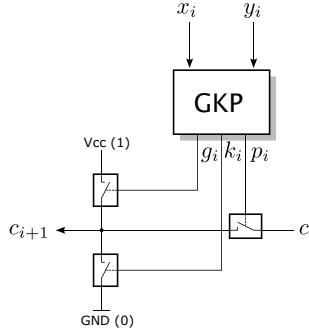


Fig. 5.11 The FA of a switched ripple-carry adder.

The benefit of this approach is very technology dependent. For instance, it was used by K. Zuse in the relay-based Z3 [Zus84; Roj97]: relays switched in parallel to compute k_i , g_i , and p_i , and this established a carry-propagation electrical circuit (see Fig. 5.11) through which the propagation of the carry itself involved no mechanical movement anymore, enabling a complete addition in one cycle. This relay-based design was later (and independently) improved by H. Aiken [Bau98].

In modern VLSI, the equivalent of a relay is the transmission gate, and a full adder cell may be designed around this idea [SB00; Bha+15]. However, transmission gates do not regenerate the signal. Therefore, this approach does not scale well to arbitrary long carry propagations (and (5.7) is not even valid in this case [AP02]).

We now look into generic methods that can be applied at the Boolean algebra level in order to speed up the carry propagation.

5.3.2 Carry-Select Adder

Another possibility to speed up the carry propagation is to split the input arguments in groups, each group computing m bits. As the input carry of each group is unknown (except for the group computing the least significant bits), the idea is to just compute both cases in parallel by using two adders and to select the correct result once the carry computation is finished. This is illustrated in Fig. 5.12 for a 16-bit adder with group size of $m = 4$ bits. The CPAs can be realized as RCAs or any other fast adder approach presented in this section. From the timing point of view, all CPAs start independent from each other. Once the carry of the first CPA is computed (c_4), it selects the correct sum bits and output carry (c_8) of the next group. This carry selects the result of the following ones, etc.

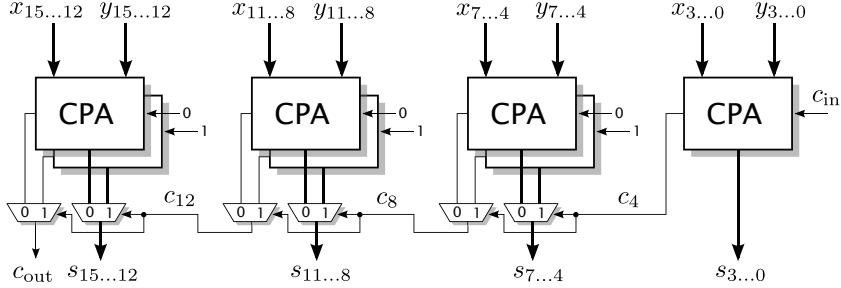


Fig. 5.12 Example of a 16 bit carry-select adder with group size of $m = 4$ bit.

The total delay is then the sum of the delay of one m -bit CPA plus the delay of $\lceil w_s / m \rceil - 1$ multiplexers required to select the carries. A slight delay improvement may be achieved by allowing for increasingly larger groups from right to left, as the leftmost groups have more time to perform their carry propagation than the rightmost ones.

5.3.3 Recursive Carry-Select Adder (Conditional-Sum Adder)

The carry-select idea can also be used recursively to build a first logarithmic-time adder ($\mathcal{O}(\log w)$ in Landau “big-O” notation [GKP94]) as follows. Assume for simplicity that w is a power of two, say $w = 2^q$. Let us build a carry-select adder with only $m = 2$ groups of size $w/2$ bits. The corresponding architecture is the rightmost half of Fig. 5.12. It consists of three adders of size $w/2$ which can operate in parallel. If we ignore for now fanout issues, the $w/2$ -bit wide multiplexer can also operate in a bit-parallel way: its delay is independent of w . The delay of the addition in this architecture can therefore be written

$$\tau_{\text{add}}(w) = \tau_{\text{add}}(w/2) + \tau_{\text{mux}} \quad . \quad (5.16)$$

Now, we may use the carry-select idea recursively for each of the three sub-adders: each is itself built as three adders of size $w/4$ followed by a multiplexer, and all these adders can operate in parallel. Thus, (5.16) becomes

$$\begin{aligned} \tau_{\text{add}}(w) &= \tau_{\text{add}}(w/4) + \tau_{\text{mux}} + \tau_{\text{mux}} \\ &= \dots \\ &= \tau_{\text{add}}(1) + q\tau_{\text{mux}} \quad . \end{aligned} \quad (5.17)$$

Since $q = \log_2 w$, we have derived a first adder architecture that computes the addition in logarithmic time with respect to the addition size. It is actu-

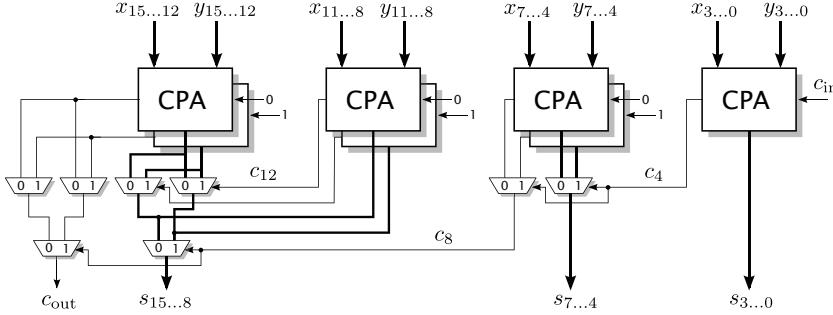


Fig. 5.13 Example of a 16 bit conditional sum adder with group size of $m = 4$ bit.

ally possible to stop the recursion earlier, typically when the adders at the leaves are of size $m = 4$ or $m = 8$.

It is important to understand that this recursive formulation does not increase the final number of CPAs compared to a carry-select adder: even though the recursive formulation seems to imply that the area is multiplied by at least $3/2$ with each level, at the leaves of the recursion, there are only two choices for the CPAs: either the input carry is 0, or it is 1. Another point of view is that a leaf CPA is shared by all the adders of the upper levels. For instance, if the recursion is stopped when the addition size is 4, we obtain an architecture with the same CPA blocks as Fig. 5.12, but with a tree of multiplexers replacing the linear sequence of multiplexers. The area thus still grows linearly with w . An architecture exploiting this observation is called a conditional-sum adder [EL04]. Figure 5.13 illustrates a 16-bit conditional-sum adder with group size of $m = 4$ bits. Note again that only the MUX routing is different compared to Fig. 5.12.

In an actual implementation of this idea, fanout issues should not be neglected: there is one carry bit that is input to a $w/2$ -bit multiplexer, hence to $w/2$ gates inside this multiplexer (e.g., c_8 on Fig. 5.13). Due to the combined electrical capacity of all these gate, for large w , this will entail a slower operation of the multiplexer. VLSI and FPGA synthesis tools are able to handle this situation automatically, but the reader should be aware that this may add some area and delay. Section 5.3.6 will present a family of adders where the fanout can be controlled.

5.3.4 Carry-Lookahead Adder

The idea of the carry-lookahead (CLA) adder is, as the name suggests, to directly compute the carry without computing intermediate carries first. For that, the recursive definition of the carry propagation

$$c_{i+1} = g_i \vee p_i c_i \quad (5.18)$$

is applied for a certain number of iterations, yielding

$$c_1 = g_0 \vee p_0 c_0 \quad (5.19)$$

$$c_2 = g_1 \vee p_1 c_1 = g_1 \vee p_1 g_0 \vee p_1 p_0 c_0 \quad (5.20)$$

$$c_3 = g_2 \vee p_2 c_2 = g_2 \vee p_2 g_1 \vee p_2 p_1 g_0 \vee p_2 p_1 p_0 c_0 \quad (5.21)$$

$$c_4 = g_3 \vee p_3 c_3 = g_3 \vee p_3 g_2 \vee p_3 p_2 g_1 \vee p_3 p_2 p_1 g_0 \vee p_3 p_2 p_1 p_0 c_0 \quad (5.22)$$

:

$$c_k = g_{k-1} \vee p_{k-1} g_{k-2} \vee p_{k-1} p_{k-2} g_{k-3} \vee \cdots \vee p_{k-1} p_{k-2} \dots p_0 c_0. \quad (5.23)$$

Now, we have a formula with just two levels of logic, one level of wide AND gates connected to a wide OR. However, the number of inputs to these gates quickly grows with k . In practice, this will introduce more levels of logic as well as a high area complexity.

One way to deal with this is to restrict the use of the lookahead scheme to a certain amount of carry propagation. Figure 5.14 shows an example of a 16-bit adder which uses groups of $m = 4$ bits. On each group, the CLA-4 module uses (5.19), (5.20), (5.21), and (5.22) for the carry computations together with (5.11) to compute the sum bits.

With this architecture, the delay still grows linearly with the word size of the adder, but with a smaller factor.

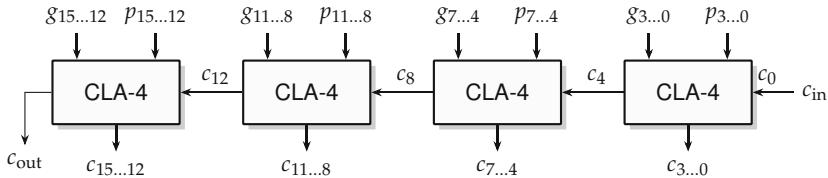


Fig. 5.14 Carry generation of a 16-bit adder with 4-bit carry-lookahead groups.

5.3.5 Recursive Carry-Lookahead Adder

The carry-lookahead idea can also be applied recursively, leading to a second family of logarithmic-time adder architectures. Here, the idea is that each CLA block of Fig. 5.14 can compute at no extra cost two additional signals $g_{i:j}$ and $p_{i:j}$, respectively, the carry generate and carry propagate of the group between indices i and j (with $i > j$), such that

$$c_{i+1} = g_{i;j} \vee p_{i;j} c_j . \quad (5.24)$$

This formulation is simply a parenthesizing of the carry-lookahead equations. For instance, (5.22) can be rewritten

$$c_4 = g_{3:0} \vee p_{3:0} c_0 \quad (5.25)$$

$$\text{with } p_{3:0} = p_3 p_2 \dots p_1 p_0 \quad (5.26)$$

$$\text{and } g_{3:0} = g_3 \vee p_3 g_2 \vee p_3 p_2 g_1 \vee p_3 p_2 p_1 g_0 . \quad (5.27)$$

These group generate and propagate signals can be computed by a tree of components similar to CLA- m . In this tree, the maximum number of levels is $\log_m(w)$, leading to an overall time complexity which scales logarithmically with w . The details can be found in [EL04].

Compared to the conditional-sum adder, we have a shallower tree – $\log_m(w)$ levels instead of $\log_2(w)$ – but with more complex nodes.

We now present a family of fast adders that generalizes the previous ideas and offers a wider implementation space.

5.3.6 Parallel Prefix Adders

The main idea here is to express the carry propagation problem as a prefix operation.

To illustrate what is a prefix operation and why it is interesting in terms of design space expressivity, let us consider a different problem: prefix sums. Assume we have a vector of n values $(X_0, X_1, \dots, X_{n-1})$ and we have to compute *all* the arithmetic sums:

$$Y_0 = X_0 \quad (5.28)$$

$$Y_1 = X_0 + X_1 \quad (5.29)$$

$$Y_2 = X_0 + X_1 + X_2 \quad (5.30)$$

$$Y_3 = X_0 + X_1 + X_2 + X_3 \quad (5.31)$$

\vdots

$$Y_{n-1} = \sum_{i=0}^{n-1} X_i . \quad (5.32)$$

Clearly, given that Y_1 is computed, one can rewrite (5.30) to $Y_2 = Y_1 + X_2$, then (5.31) to $Y_3 = Y_2 + X_3$, etc. This is a resource-efficient way to compute all the prefixes, but also the slowest as it is completely sequential, with a succession of n additions. Figure 5.15a illustrates this computation scheme for the sum of $n = 8$ values.

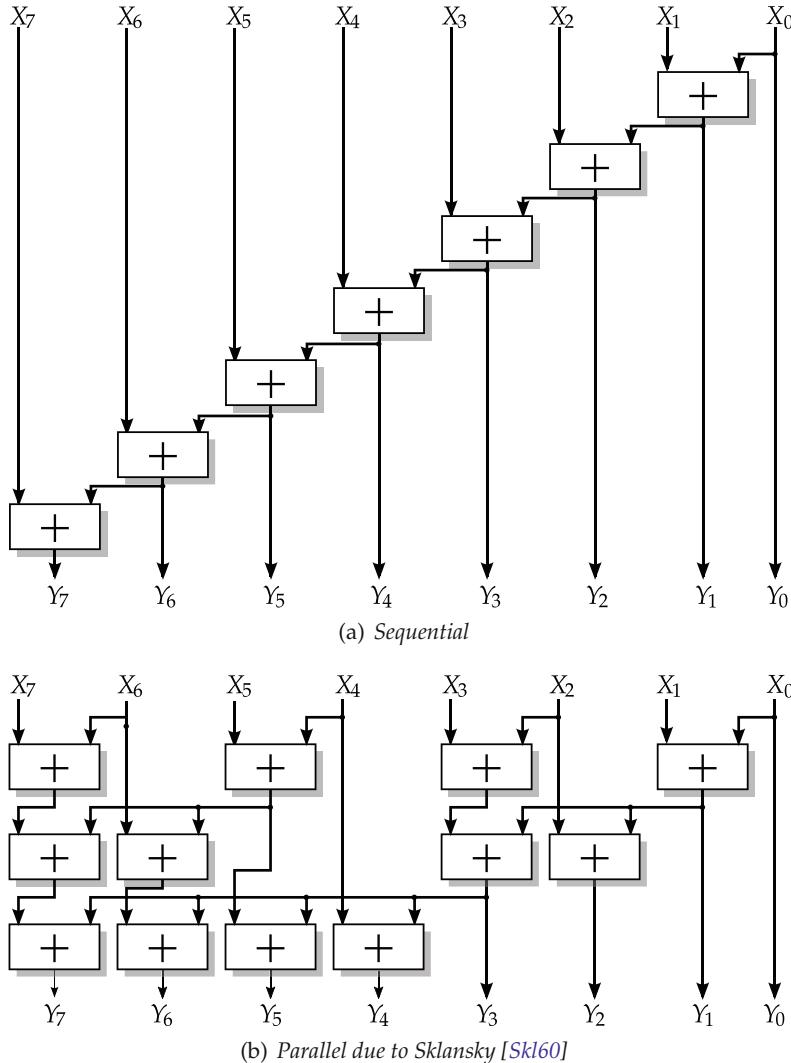


Fig. 5.15 Examples of computations of a parallel prefix sum.

On the contrary, if the objective is to achieve the maximum speed, one can build one adder tree for each of the outputs. All these adder trees will work in parallel, which guarantees that all the computations will be achieved in $\log_2(n)$ additions. However, the hardware cost will be high. One way to reduce it is to exploit the associativity of addition: many intermediate results can be shared while still achieving a worst-case $\log_2(n)$ time, as illustrated in Fig. 5.15b [Skl60]. The reader may check that Fig. 5.15b still computes the same as Fig. 5.15a, with a moderate increase in the number of adders (12 adders in Fig. 5.15b compared to 8 adders of Fig. 5.15a).

We have exposed a trade-off between time and area. This problem is an instance of a *parallel prefix operation*, where the objective is to compute all the prefixes according to some associative operation, here $+$. There are actually many other intermediate solutions with different trade-offs; see [Zim97] for an extensive review including cost evaluations for area, time, and area-time product.

Back to the subject of this chapter, let us show that carry propagation can also be expressed as a prefix computation, for an operation \circ defined on 2-bit tuples formed of a generate bit and an alive³ bit (see (5.10))

$$X = (g_X, a_X) \quad (5.33)$$

$$Y = (g_Y, a_Y) \quad (5.34)$$

which are related by a prefix operation

$$Z = (g_Z, a_Z) = X \circ Y = (g_X, a_X) \circ (g_Y, a_Y) \quad (5.35)$$

with

$$g_Z = g_X \vee a_X g_Y \quad (5.36)$$

$$a_Z = a_X a_Y. \quad (5.37)$$

This operation is associative, i.e.,

$$(U \circ V) \circ W = U \circ (V \circ W) \quad (5.38)$$

which can be easily proven by setting (5.36) and (5.37) into the left- and right-hand side of (5.38) and showing equality:

$$g_{(U \circ V) \circ W} = g_{U \circ V} \vee a_{U \circ V} g_W = g_U \vee a_U g_V \vee a_U a_V g_W \quad (5.39)$$

$$g_{U \circ (V \circ W)} = g_U \vee a_U g_{V \circ W} = g_U \vee a_U (g_V \vee a_V g_W) \quad (5.40)$$

$$= g_U \vee a_U g_V \vee a_U a_V g_W \quad (5.41)$$

$$a_{(U \circ V) \circ W} = a_{U \circ V} a_W = a_U a_V a_W \quad (5.42)$$

$$a_{U \circ (V \circ W)} = a_U a_{V \circ W} = a_U a_V a_W. \quad (5.43)$$

With this operation, we can obtain the carries by computing

³ It is also possible to use the generate and propagate bits, e.g., in [Zim97].

$$(g_0, a_0) \circ (c_0, c_0) = (g_0 \vee a_0 c_0, a_0 c_0) = (c_1, a_0 c_0) \quad (5.44)$$

$$(g_1, a_1) \circ (c_1, a_0 c_0) = (g_1 \vee a_1 c_1, a_1 a_0 c_0) = (c_2, a_1 a_0 c_0) \quad (5.45)$$

$$(g_2, a_2) \circ (c_2, a_1 a_0 c_0) = (g_2 \vee a_2 c_2, a_2 a_1 a_0 c_0) = (c_3, a_2 a_1 a_0 c_0) \quad (5.46)$$

$$\vdots$$

$$(g_{i-1}, a_{i-1}) \circ (c_{i-1}, a_{i-1} a_{i-2} \dots a_0 c_0) = (c_i, a_{i-1} a_{i-2} \dots a_0 c_0). \quad (5.47)$$

So, the carry at position i can be computed by evaluating

$$(c_i, a_{i-1} a_{i-2} \dots a_0 c_0) = \bigcirc_{j=-1}^{i-1} (g_j, a_j) \quad (5.48)$$

when defining $g_{-1} = a_{-1} = c_0$.

Hence, as we need all the carries, we can perform the same rearrangement done for computing the sums in the example above. This is shown in Figs. 5.17, 5.18, and 5.19. In the figures, two different notations are used which are introduced in Fig. 5.16. The reason is that in some prefix additions, no a_R is given and thus no a_{out} result is computed. Therefore we can omit its computation. Those simplified prefix computations are illustrated using a filled circle⁴ (see Fig. 5.16).

The obtained carries have to be XORed with the propagate signal to obtain the final sum. These XORs are not shown on the figures.

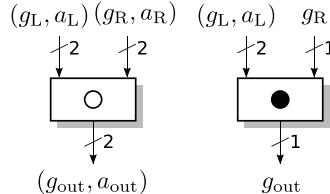


Fig. 5.16 The prefix operator and its simplified version.

The only rule to consider when organizing the prefix operators is (5.48). This allows many other variants with different properties. Regarding the delay, the stage count is the main metric to consider, but another important one is the *fanout* of each operator. The fanout of an output counts the number of inputs that are connected to this output. Due to the combined electrical capacity of these inputs, a high fanout entails slow transitions. This can be partially compensated in very large scale integrated circuit (VLSI) circuits by using a stronger driver, but this also leads to a larger power consumption. Hence, designs with low fanout are desirable. Figure 5.19 shows the parallel prefix adder according to Kogge and Stone [KS73], which limits the fanout to two (compared to $w/2$ in the Sklansky structure of Fig. 5.18) while still

⁴ Different notations are used in the literature; we use the notation of [EL04] here while, e.g., [Zim97] uses a different notation.

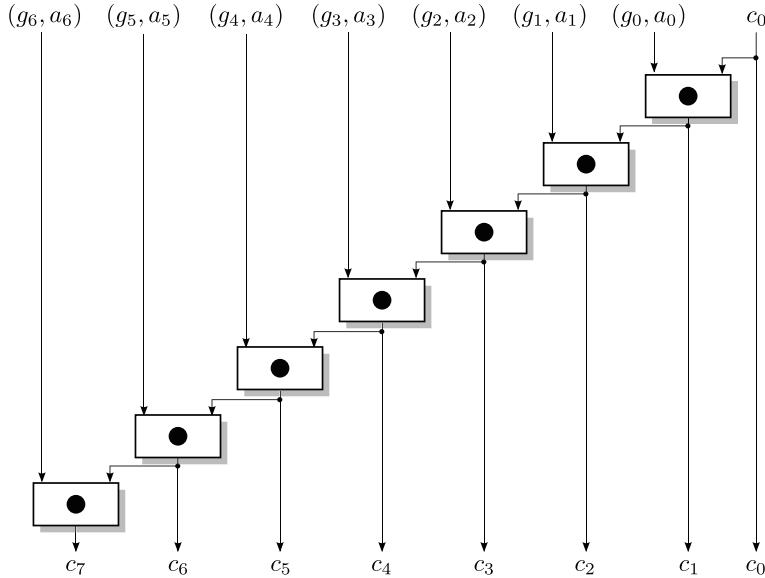


Fig. 5.17 The carry-propagate adder as a special case of parallel-prefix adder.

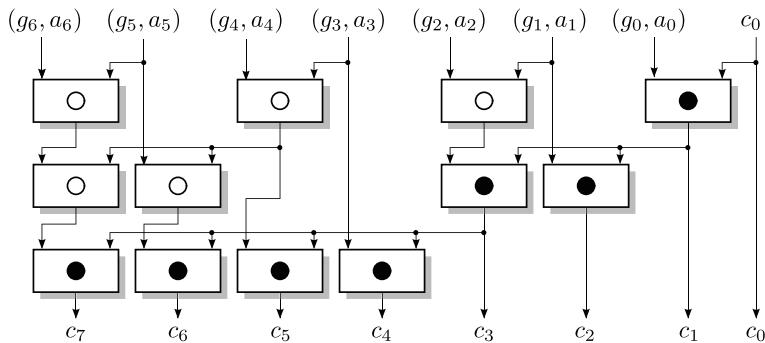


Fig. 5.18 The Sklansky parallel-prefix adder.

providing the minimum number of stages. This comes at the cost of more resources than the Sklansky structure.

Another example is the Brent-Kung adder [BK1], which uses very low resources while having a slightly larger stage count. Generalizations have also been made toward prefix operators accepting more than two inputs [BL01].

An excellent review of various prefix structures comparing area, delay, wire cost, and fanout is given in Zimmermann's PhD thesis [Zim97]. Another good overview, including a taxonomy comparing the stage count (logic depth), the fanout, as well as the complexity of the wire tracks was presented by Harris [Har03]. The issue of power consumption in fast adders has been studied by Patil et al. [Pat+07]. To address the huge design space

of parallel prefix adders, an NVIDIA team has used reinforcement learning [Roy+21]. Prefix adders are also well suited to adders with specific timing requirements, i.e., input bits arriving at different times or output bits having different timing constraints [Zim97, Chapter 5].

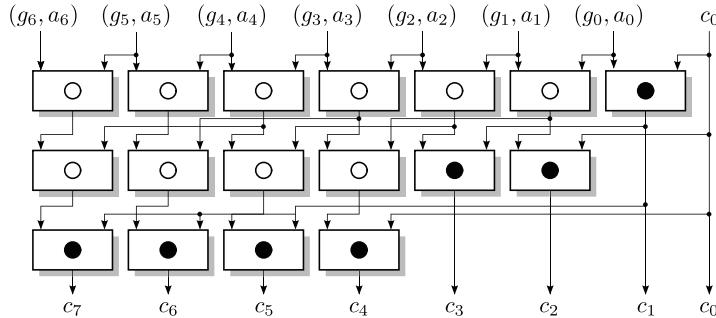


Fig. 5.19 A Kogge-Stone parallel prefix adder.

5.3.7 Compound Fast Adder

In a parallel prefix adder, the bulk of the operator computes all the immediate generate and alive signals. The computation that actually depends on c_0 , the input carry, is only the bottom line of simplified prefix operators (see Figs. 5.18 and 5.19). Its size is linear in w ; therefore it is relatively cheap to duplicate it for $c_0 = 0$ and $c_0 = 1$. This way, a fast parallel prefix adder computing $X + Y$ may also compute $X + Y + 1$ for little additional cost, as illustrated in Fig. 5.20.

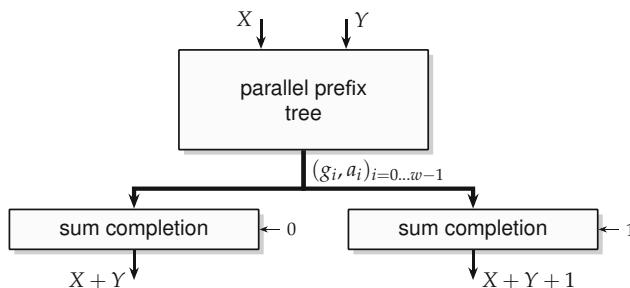


Fig. 5.20 Compound adder computing both $X + Y$ and $X + Y + 1$.

Such a *compound adder* is very useful for floating-point operators, where the rounded significand is either a sum or the successor of this sum [SE01].

5.3.8 Fast Absolute Difference Operator

A variation of the compound adder, depicted in Fig. 5.21, computes $|X - Y|$ when X and Y are two positive numbers. With two's complement, this operator needs to compute either $X - Y = X + \bar{Y} + 1$ or $Y - X = \bar{X} + \bar{Y}$.⁵ The selection is done in constant time by a multiplexer controlled by the sign bit of one of the differences ($Y - X$ on our figure). The two bitwise negations in these equations do not necessarily entail additional delay: \bar{Y} can be merged in the initial (g_i, a_i) computation, and the last negation of $\bar{X} + \bar{Y}$ can be merged in the sum completion logic.

Note that the operator of Fig. 5.21 may also output (for free) the sign bit of $Y - X$, or the sign bit of $X - Y$, or both. The two sign bits will be different, except when $X = Y$: at the cost of an additional NOR gate, this operator may also provide an $X = Y$ bit.

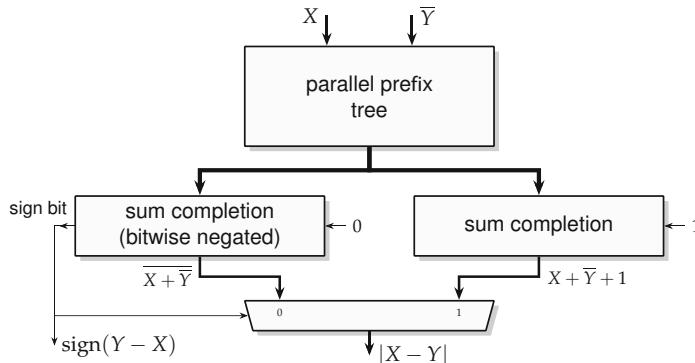


Fig. 5.21 Compound adder computing $|X - Y|$.

5.3.9 Fast Compound Triple Sum

Some fast floating-point adders [Soh+22] require the computation of $X + Y$, $X + Y + 1$, and $X + Y + 2$ in parallel. Here is a simple way to do implement

⁵ Proof that $Y - X = \bar{X} + \bar{Y}$: we have $X - Y = X + \bar{Y} + 1$, hence, $X - Y - 1 = X + \bar{Y}$; the opposite of this value is $-(X - Y - 1) = \bar{X} + \bar{Y} + 1$ hence $Y - X = \bar{X} + \bar{Y}$.

this [Soh+22]. First, the two addends X and Y are input into a row of half adders (HAs) that rewrite the sum $X + Y$ into sum and carry vectors $S + C$ (Fig. 5.22). The advantage of this rewriting is that the sum bit at position 0 is reduced to a single bit s_0 .

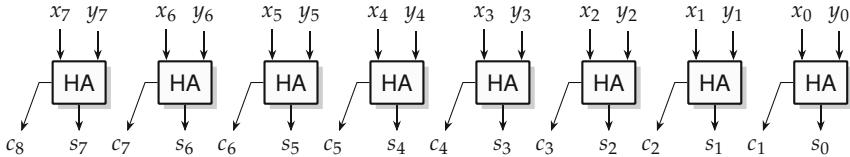


Fig. 5.22 Rewriting $X + Y = S + C$ using a row of half adders.

Then, a compound adder on the upper bits (positions 1 to $w + 1$, i.e., assuming $s_0 = 0$) computes two intermediate sums Z and $Z' = Z + 2$. With this, the triple sum can be expressed (at the cost of a multiplexer) as the following concatenations:

$$\text{if } s_0 = 0 \begin{cases} X + Y = Z0 \\ X + Y + 1 = Z1 \\ X + Y + 2 = Z'0 \end{cases} \quad \text{else} \quad \begin{cases} X + Y = Z1 \\ X + Y + 1 = Z'0 \\ X + Y + 2 = Z'1 \end{cases} . \quad (5.49)$$

In short, the overhead of the fast triple adder with respect to the compound fast adder is a row of w half adders and three $w + 2$ -bit multiplexers.

5.4 Adders on FPGAs

Fixed-point adders on FPGAs have two particularities that are worth studying. The first is that carry propagation benefits from dedicated hardened (i.e., non-programmable) hardware and routing resources (collectively named *fast carry logic*; see Sect. 4.1, p. 91) which make it very fast compared to generic programmable logic. The second is that additions can be merged with other functionalities at no additional cost. This section studies these two points in more details for the two mainstream FPGA families from AMD and Intel.

5.4.1 Addition Support on AMD FPGAs

On AMD field programmable gate arrays (FPGAs), the adder implementation depends on the series. On series 4 to 7 and UltraScale/UltraScale+ FPGAs, the carry chain logic consists of an XOR gate, and a MUX per basic logic element (BLE) (see Fig. 4.5b in Sect. 4.1.2, p. 91) can be used to build ripple-carry adders (RCAs). For this, the look-up table (LUT) has to be configured to compute the propagate signal $p_i = x_i \oplus y_i$. The sum output is computed by XORing p_i with c_i with the carry chain XOR. The carry computation uses the MUX formulation obtained in (5.15). Hence, each BLE computes the Boolean relations

$$s_i = p_i \oplus c_i \quad (5.50)$$

$$c_{i+1} = p_i c_i \vee \overline{p_i} x_i \quad (5.51)$$

which correspond to the full adder (FA) relations (5.2) and (5.15). The BLE configuration is shown in Fig. 5.23.

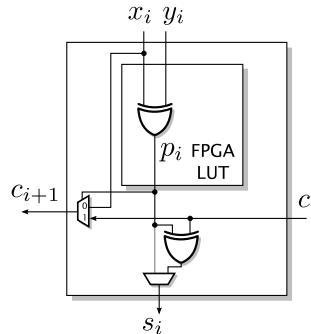


Fig. 5.23 Realization of a full adder on Xilinx/AMD's series 4 to 7 and UltraScale/UltraScale+.

It can also be seen as a switched ripple-carry adder (see Sect. 5.3.1) where the XOR in the LUT computes the propagate signal p_i . Now, if $p_i = 1$, the carry in is propagated to the output. Otherwise, if $p_i = 0$, it is sufficient to pass either x_i or y_i , as it is known that both are equal (as their XOR obtained zero due to $p_i = 0$).

This multiplexer is very fast compared to the surrounding logic, which is why this is called the *fast carry chain*. To give some example values, consider the timing data of the Virtex 6 FPGA architecture (similar ratios can be assumed for other families). It comes in different performance grades, which are called *speed grades*. For the fastest speed grade (3), the carry-in to carry-out delay of a complete slice with four BLEs is specified to be at most 0.06 ns [Xil14]. Hence, each BLE contributes with $\tau_{cp} = 0.015$ ns to the carry delay. The relevant combinational delay of a LUT ranges between

$\tau_{\text{LUT}} = 0.14 \dots 0.32 \text{ ns}$, depending which input influences which output [Xil14]. The delay of the XOR (from carry-in to one of the BLE outputs within a slice) is $\tau_{\text{XOR}} = 0.21 \dots 0.25 \text{ ns}$. This leads to a worst-case full adder delay of $\tau_{\text{fa}} = 0.57$, which is $38 \times$ the delay of the carry chain. Besides, there is also a dedicated link for the transmission of a carry bit from one CLB to the next, which is also much faster than random wiring through the generic routing fabric. The worst-case delays of adders with $w = 8$, $w = 16$, and $w = 32$ bit can be obtained using (5.7) and are about 1, 1.5, and 2.5 ns, respectively.

On the Versal devices, the carry chain logic of each BLE does not contain the XOR gate anymore, but an equivalent of the MUX is included in the fast carry-lookahead logic (see Fig. 4.5c in Sect. 4.1.2, p. 91). The fast carry-lookahead logic computes every second carry from all the propagate and intermediate carries computed in the LUTs. From the eight BLEs in a slice which are labeled from A to H, but only the even carry outputs COUTB, COUTD, COUTF, and COUTH are computed from the carry logic in a lookahead manner. In consequence, the remaining odd carry outputs have to be computed by using the LUTs.

Figure 5.24 shows the configuration of three BLEs together with the carry-lookahead logic at the bottom. The propagate signals p_i are computed using LUTs; in this case, each BLE computes a propagate in the LUT4 that is connected to the PROSSP input of the carry-lookahead logic, called the LOOKAHEAD8 unit. As no XOR is available in the carry chain anymore, we have to utilize the LUTs. For that, the carry c_i of the neighboring BLE is routed by casc_{in} to the MUX that corresponds to O5_1. With that, sum signal is computed by

$$s_i = p_i \bar{c}_i \vee \bar{p}_i c_i \quad (5.52)$$

$$= p_i \oplus c_i, \quad (5.53)$$

where \bar{p}_i is computed in the second LUT4.

For BLEs that compute inputs with even indices (c_0, c_2, \dots), the carry is either given as input (c_0) or computed by the carry-lookahead logic (c_2, c_4, \dots) and provided at the casc_{in} of the BLE. For BLEs that compute inputs with odd indices (c_1, c_3, \dots), the carry has to be computed in the previous BLE and is passed by casc_{in} of the BLE. Hence, the odd indexed carry outputs are computed by

$$c_{i+1} = x_i y_i \bar{c}_i \vee (x_i \vee y_i) c_i \quad (5.54)$$

$$= x_i y_i \bar{c}_i \vee x_i c_i \vee y_i c_i \quad (5.55)$$

$$= x_i y_i \vee x_i c_i \vee y_i c_i. \quad (5.56)$$

This is realized in the remaining two LUT4 and outputted at O5_2, O6, and finally casc_{out}. In the even indexed BLEs, one of the inputs have to be passed to the carry-lookahead logic according to (5.15). With that, we will only have a carry propagation from even indexed BLEs to odd indexed BLEs. Together

with the carry-lookahead scheme of the LOOKAHEAD8 unit, this will result in fast adders that are very flexible in size.

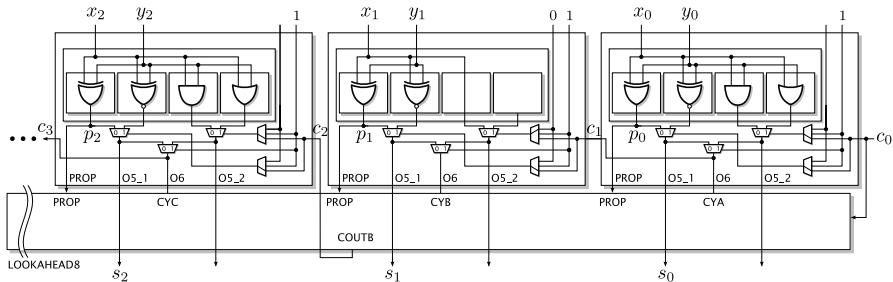


Fig. 5.24 Realization of the first bits of an adder using AMD's Versal BLEs.

All this is exploited transparently by synthesis tools: the take-away message here is that an FPGA designer should not worry too much about the carry propagation delay of additions of sizes up to 32-bit. However, another important remark is that the addition logic is not fully using the BLE. We will shortly see (Sect. 5.4.3 and the subsequent sections) how this remark can be exploited.

5.4.2 Addition Support on Intel FPGAs

Things are simpler on Intel hardware, as each BLE includes a complete FA with dedicated carry propagation wires to its neighbors (see Sect. 4.1.3). The implementation of a RCA is straightforward. There are only a few constraints on the start and stop of the carry chain: there are ten adaptative logic modules (ALMs) per logic array block (LAB) (each ALM consisting of two BLEs with partially shared inputs). A carry chain can start at the first or the sixth ALM of an LAB.

Here also, carry propagation is very fast compared to generic programmable logic thanks to the fact that these FA are hardened and that the generic programmable routing is avoided.

5.4.3 Merging Additional Functionality in an Adder

Implementing adders on FPGAs will only use a fraction of the LUT resources. On Intel FPGAs, the inputs of the FA can be fed from the LUTs (see Fig. 4.6, p. 93), which can be used for additional logic without additional cost. This enables an RCA computing

$$(c_{i+1}, s) = \text{FA}(p(a, b, c, d), q(b, c, d, e), c_i), \quad (5.57)$$

using the FA notation introduced in (5.4) where p and q are two bit-parallel functions implemented as parallel LUT4 sharing most of the inputs.

We have a similar case for AMD FPGAs as only a fraction of the LUT is utilized in Fig. 5.23. Hence, the remaining logic can be used for additional logic at no cost. The BLE of Fig. 4.5b can realize two independent LUT5 with shared inputs in front of each of the FA inputs computing

$$(c_{i+1}, s) = \text{FA}(p(a, b, c, d, e), q(a, b, c, d, e), c_i) \text{ (Fig. 5.25a).} \quad (5.58)$$

Here, two parallel LUT5 with shared inputs are possible. Alternatively, there can be a single LUT6 on one of the FA inputs, allowing to perform

$$(c_{i+1}, s) = \text{FA}(p(a, b, c, d, e, f), a, c_i) \text{ (Fig. 5.25b).} \quad (5.59)$$

The BLE of the Versal architecture in Fig. 4.5c can realize two independent LUT4 with shared inputs in front of each of the FA inputs computing

$$(c_{i+1}, s) = \text{FA}(p(a, b, c, d), q(a, b, c, d), c_i) \text{ (Fig. 5.25c).} \quad (5.60)$$

This additional logic can be used, e.g., to realize

- an additional full adder adding a third input (see Sect. 5.4.4),
- an adder/subtractor, where one input bit selects between addition and subtraction at runtime (see Sect. 5.4.5),
- a multiplexer selecting between different sources for the addition (see, e.g., Fig. 9.8 in the dividers of Sect. 9.2.4),
- the addition of a value read from a table indexed with fewer than 5 bits inputs (see, e.g., table-based constant multiplications [Wir04] in Sect. 12.2.2 or optimizations to an arctangent implementation [TVC17]),
- the generation of partial products in multiplications (see Sect. 8.4.2).

This is one example of *target-specific optimization*, one of the techniques for application-specific arithmetic discussed in the introduction.

The remainder of this section gives two examples where additional functionality is merged in the LUTs implementing the addition: Sect. 5.4.4 shows the construction of a ternary adder, and Sect. 5.4.5 discusses the construction of several adder/subtractor variants.

5.4.4 Ternary Adders on FPGAs

The logic available for free in front of the RCA can be used to realize additional full adders to be able to add a third number with essentially the same

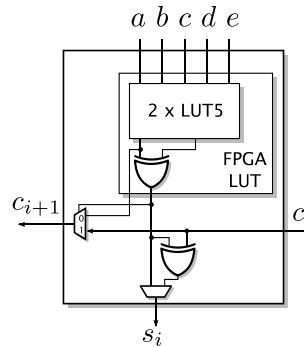
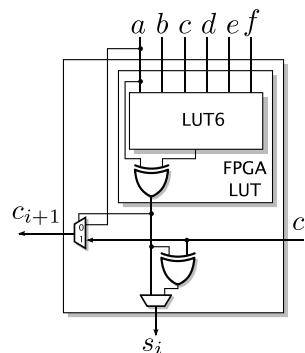
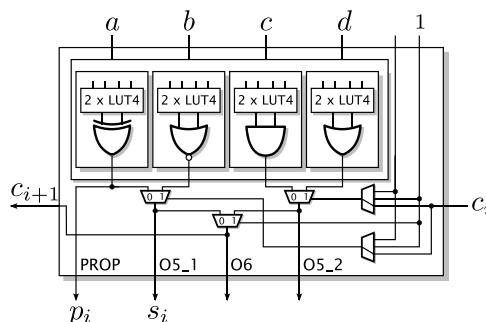
(a) FA plus $2 \times \text{LUT5}$ in Virtex/Ultrascale(+)(b) FA plus $1 \times \text{LUT6}$ in Virtex/Ultrascale(+)(c) FA plus $1 \times \text{LUT6}$ in Versal

Fig. 5.25 Additional logic that can be implemented in front of a full adder using AMD's BLEs (see Fig. 4.5b and c).

logic resources as for common 2-input adders. Hence, three variables can be added in one row of BLEs, computing

$$S = X + Y + Z. \quad (5.61)$$

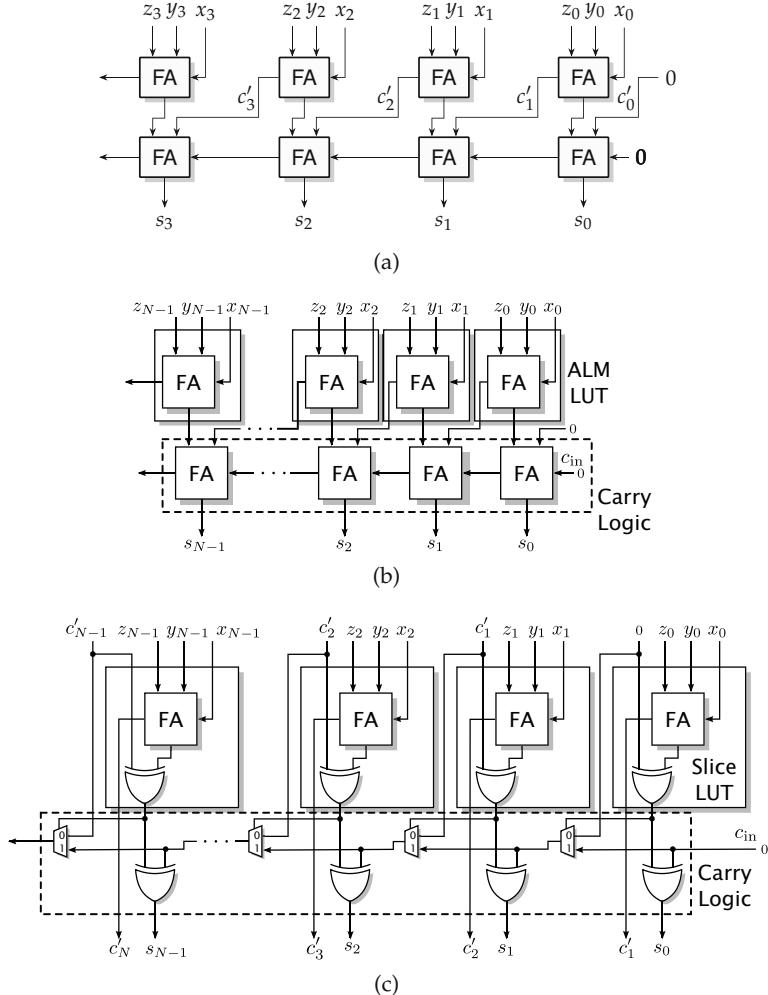


Fig. 5.26 Realization of ternary adders (a) generic, (b) mapping to Intel Stratix II-V ALMs (c) mapping to Xilinx/AMD Virtex 5-7 slices.

These adders are commonly called ternary adders (not to be confused with adders using a ternary *number format*, i.e., $\{-1, 0, 1\}$ [EL04]).

The ternary adder is realized by implementing one layer of full adders to form a carry-save adder (CSA) which compresses the three input vectors into two vectors (sum vector and carry vector). More details about carry-save arithmetic can be found in Chap. 7. These two vectors are added to get the final sum by using an RCA built from the fast carry chain. This principle is shown in Fig. 5.26a. The first row of full adders realizes the CSA; the second row of full adders corresponds to the RCA. Note that there is no

carry-propagation in the CSA. In VLSI, it would not save any hardware as the same number of FAs is used. However, on FPGAs the first layer of FAs can be mapped to the LUTs in front of the RCA. The detailed implementation for the different FPGA vendors are briefly given in the following.

5.4.4.1 Realization on Intel FPGAs

The ALMs of Intels Arria I, II, and V and Stratix II-V FPGAs support a special *shared arithmetic mode* for the fast implementation of ternary adders [Bae+09; Alt13]. Unfortunately, the shared arithmetic mode has been dropped with the Stratix 10 generation [Int17; Int18a]. For FPGAs supporting the shared arithmetic mode, the FAs of the CSA can be directly mapped to the LUTs as shown in Fig. 5.26b. Note that a 2-input adder with the same output word size requires exactly the same number of ALMs, but will be slightly faster [Kum+13]. For FPGAs not supporting the shared arithmetic mode, the ternary adder requires additional LUTs [Int18a].

The Quartus II tool automatically detects ternary adders from a VHDL statement of the form $S \leftarrow X + Y + Z$. Unfortunately, ternary operations when one or more inputs are negated (i.e., $X - Y + Z$, $X + Y - Z$, and $X - Y - Z$) are not directly supported from a high-level description. To overcome this problem, the corresponding input(s) must be inverted, and the carry-in signal(s) must be set to 1. To set the carry bit(s), the word size of the adder has to be extended by one bit for $X - Y + Z$ and $X + Y - Z$ and by two bits for $X - Y - Z$, consuming additional ALMs.

5.4.4.2 Realization on AMD FPGAs

Ternary adders can be efficiently mapped to all modern Xilinx/AMD FPGA families where the BLE provides a 6-input LUT that can be fractioned as two 5-input LUTs [SP06], namely, the Virtex 5-7, Spartan 6, Kintex 7, Artix 7 families, as well as the Zync System-on-Chips (SoCs) (including UltraScale and UltraScale+).

The slice configuration is shown in Fig. 5.26c. The full adder for the CSA may be realized in the same LUT as the XOR gate of Fig. 5.23. The carry output c'_i are all computed in parallel, but unfortunately they have to be routed to the next higher FA through the FPGA routing fabric. As a result, for low word sizes [Kum+13], a ternary adder is up to 50% slower than a binary one. Still, the carry propagation of the second adder only uses the fast carry dedicated routing (dashed box of Fig. 5.26c).

Currently, there is no guarantee that ternary adders from an hardware description language (HDL) description are mapped in that efficient way by the AMD tools. Thus, it has to be built by using the AMD primitives. Note

that the ternary subtract operations ($X - Y + Z$, $X + Y - Z$ and $X - Y - Z$) can be realized without additional resources.

Hands on: Ternary adder for AMD/Xilinx FPGAs

The FloPoCo core generator can be used to build ternary adders for AMD/Xilinx FPGAs. The following FloPoCo call generates VHDL code for a ternary adder with 16 bit input word size:

```
flopoco target=virtex6 XilinxTernaryAddSub wIn=16
```

An optional bit mask can be passed to select inputs to be subtracted (with the LSB corresponding to input X and the MSB corresponding to Z). For instance, the bit mask $011_2 = 3_{10}$ selects the $-X - Y + Z$ operation:

```
flopoco target=virtex6 XilinxTernaryAddSub wIn=16 \
    AddSubBitMask=3
```

5.4.5 Reconfigurable Adder/Subtractor

The structure of an adder/subtractor as introduced in Sect. 5.2.5 is easy to map to the current FPGA architectures. It comes at no additional cost compared to a simple adder or subtractor as the required XORs can be mapped to the same LUTs used for the RCA.

The idea can be even further extended such that both inputs may be negated. Clearly, adding a select signal sub_X which is XORed with the X input and ORed with the carry-input would also allow a selective $-X + Y$. However, to realize $-X - Y$, one would have to add constant $1 + 1 = 2$ for complementing both inputs. This is, of course, not possible just by using the carry-in. However, the structure of the ternary adder described above can be used to perform this [KKZ16]. Such a generalized adder computes

$$S = \begin{cases} X + Y & \text{when } \text{sub}_X = 0, \text{sub}_Y = 0 \\ X - Y & \text{when } \text{sub}_X = 0, \text{sub}_Y = 1 \\ -X + Y & \text{when } \text{sub}_X = 1, \text{sub}_Y = 0 \\ -X - Y & \text{when } \text{sub}_X = 1, \text{sub}_Y = 1 \end{cases}. \quad (5.62)$$

Its FPGA mappings to Intel and AMD FPGAs are shown in Figs. 5.27 and 5.28, respectively. Only when exactly one of the select inputs is negated, a $+1$ has to be added which is realized by an XOR gate. The $+2$ is added in the case that both select inputs are true, which is realized by an AND gate.

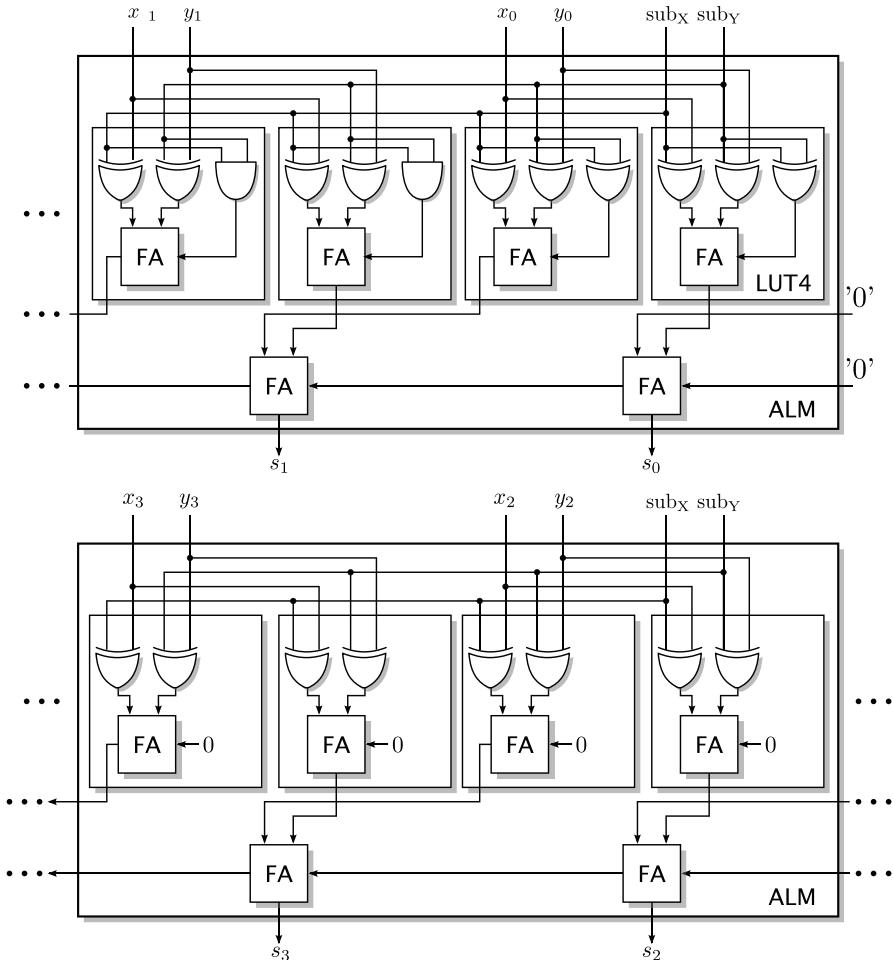


Fig. 5.27 Intel ALM mapping of the adder/subtractor computing $S = (-1)^{\text{sub}_X} X + (-1)^{\text{sub}_Y} Y$.

This generalized adder/subtractor is especially interesting for adding up absolute values (by using the sign bit of X and Y as sub_X and sub_Y , respectively). This is, for example, used in the sum of absolute difference (SAD) metric widely found in image processing [KKZ16].

5.5 Fast Adders on FPGAs

Fast adder architectures as discussed in Sect. 5.3 are useless when targeting FPGAs for typical sizes (up to 32 bits). However, several FPGA-specific

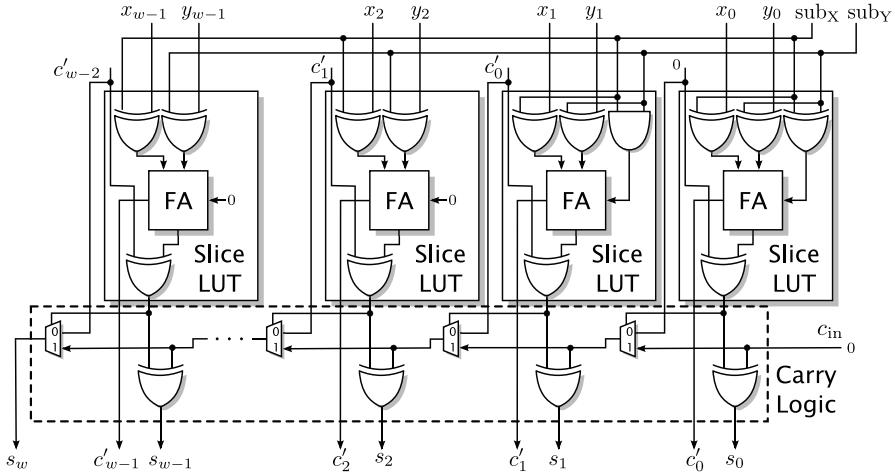


Fig. 5.28 AMD Slice mapping of the adder/subtractor computing $S = (-1)^{\text{sub}X} X + (-1)^{\text{sub}Y} Y$.

schemes have been suggested to accelerate larger additions on FPGAs [XY98; YN08; DNP10; NPP11; RHG14; KG16; LPB19]. Accelerating fast addition is an active research topic, as there are plenty of applications for very large word sizes, in particular in cryptography [RHG14; LPB19].

There are in principle two ways to accelerate the speed: pipelining and optimizations of the combinatorial paths (like discussed above in Sect. 5.3 for VLSI adders). Of course, both can be combined. They are discussed in the following.

5.5.1 Pipelined Adders

In VLSI, the cost of a flip flop is roughly comparable to that of a full adder; therefore pipelining an integer addition is not competitive compared to the fast adder techniques of Sect. 5.3 (all the more as pipelining delays the output by one or several cycles, which may incur more flip-flops (FFs) for synchronization of the output with other signals). Conversely, FPGAs offer a LUT/FF ratio close to 1: FFs are a resource that is wasted if it is not used. Besides, FFs are integrated in the BLEs and directly connected to the addition output (see Sect. 4.1), so pipelining is an attractive way to speed up integer addition on FPGAs.

Figure 5.29a shows an RCA after applying classic pipelining. The RCA is divided into groups of m bits, and each group is computed in a separate pipeline stage. Hence, each pipeline stage except stage 0 requires $2m$ FFs for the inputs, m FFs for the output, and a single FF for the carry. Note that this

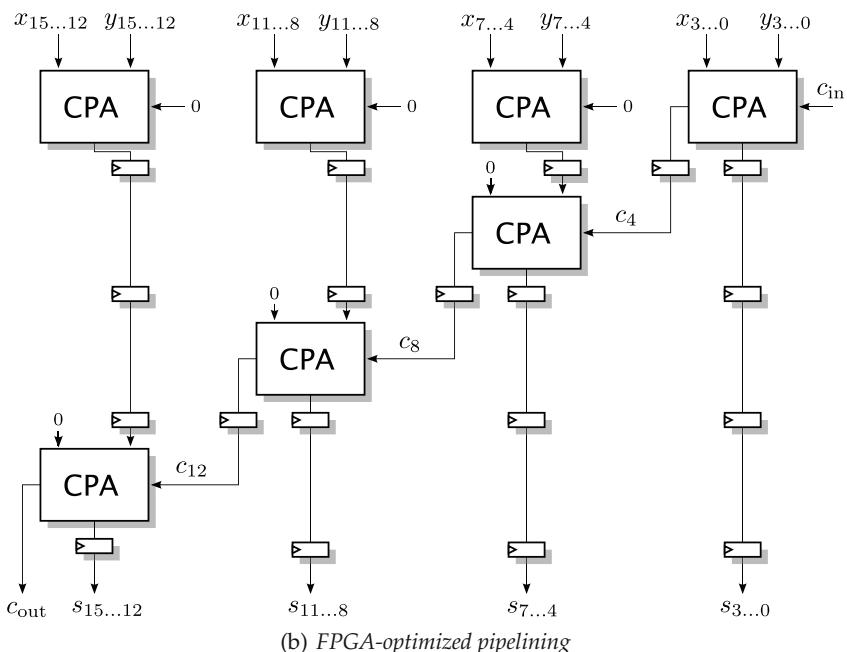
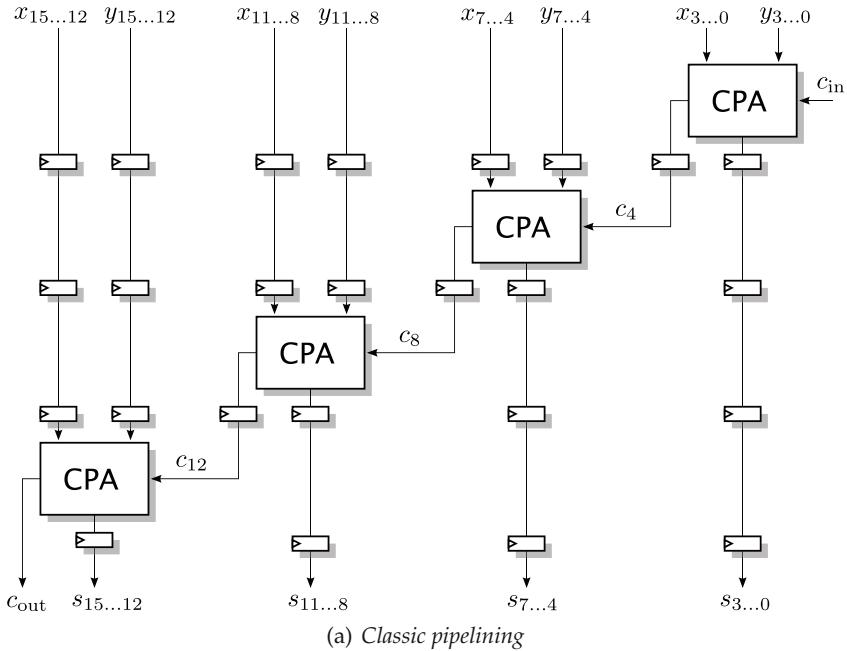


Fig. 5.29 Pipelined ripple-carry adder with 16 bits and a group size of $m = 4$ bit.

FF is the only one that actually cuts the critical path; the other $3m$ FFs are just to balance the pipeline.

Here, all of the output registers use the registers of the BLEs computing the addition and can be considered to come for free. However, the registers for balancing the inputs come at a cost: they have to come from other BLEs. In case of several FFs in a sequence, they may use the shift register LUT (SRL) as discussed in Sect. 4.3.2. In any case, the number of BLEs for pipeline balancing may be higher than the BLEs required for the actual summation.

Figure 5.29b shows a solution presented in [DNP10] that avoids about half of the BLEs required for input balancing. It uses the fact that in terms of BLE consumption, a FF has a similar cost as a FF + FA. Hence, m bit adders are used in the first stage to compress the amount of bits to store within the following pipeline stages. Then, the following adders are just used to add the carries of the previous stage.

The pipelined adder is a very generic way to speed up adders to any word size at a reasonable cost, but at the expense of a potentially high latency. To reduce this latency, further logic can be used to reduce the delay.

Hands on: Pipelined adders

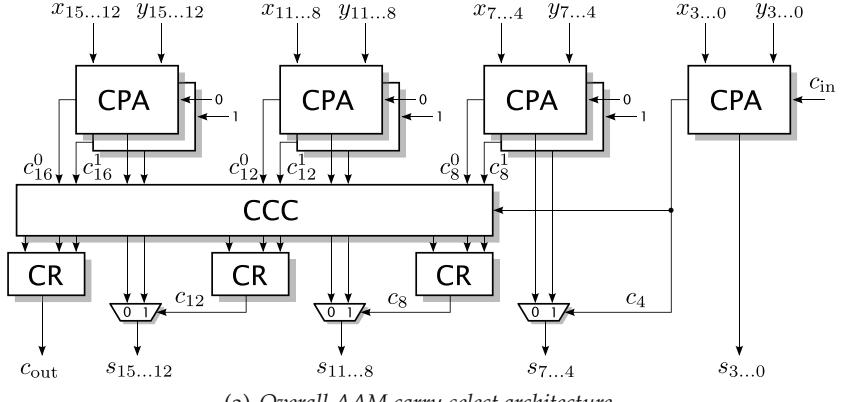
The FloPoCo core generator can be used to build pipelined adders that meet a given target frequency. The following FloPoCo call will generate a pipelined adder according to Fig. 5.29a with 1024 bit input word size:

```
flopoco frequency=300 IntAdder wIn=1024
```

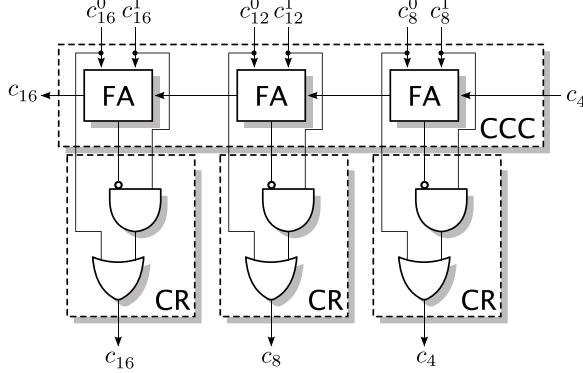
The reader is invited to play with the optional parameters of `IntAdder` and also with the `target` and `frequency` parameters to observe their impact on the group size and pipeline depth.

5.5.2 Fast Combinatorial Adders

One method for reducing the delay was first presented in [DNP10] and later improved in [NPP11; RHG14; LPB19]. We present the idea of the AAM carry-select architecture of [NPP11] in the following. This architecture basically applies the carry-select idea, but uses a different selection scheme. The architecture is shown in Fig. 5.30a for the example of a 16-bit adder split into groups of $m = 4$ bits. Note that this is a toy size for explaining the concept, as a delay benefit will only occur for much larger sizes (256 bits or more [NPP11]).



(a) Overall AAM carry-select architecture.



(b) Carry computation circuit (CCC) with carry recovery (CR).

Fig. 5.30 add-add multiplex (AAM) carry-select architecture for a 16-bit adder.

Like in the carry-select scheme (see Sect. 5.3.2), two adders are applied per segment in the first stage, computing both possible results for carry-in equals 0 and 1 in parallel. The resulting carry candidates c_i^0 and c_i^1 of the first stage are now used to select the correct carries. In contrast to the classic approach, this is not performed by MUXes but by using fast prefix computations that can be efficiently mapped to the fast carry chain. These are shown as a carry computation circuit (CCC) block in Fig. 5.30a and a small LUT-based carry recovery (CR) circuit.

The CCC as well as the CR circuits are shown in Fig. 5.30b. The CCC is nothing less than a ripple-carry adder. Table 5.2 shows the truth table how the actual carry c_i is computed from both carry candidates c_i^0 and c_i^1 computed in the first stage and the carry of the previous group c_{i-m} . The same cases like in a full adder occur, except that the propagate case $c_i^0 c_i^1$ is obviously impossible. Hence, the output carry c_i can be computed by

Table 5.2 Cases in carry-propagation in the carry-select scheme.

c_i^0	c_i^1	c_i	Case
0	0	0	kill
0	1	c_{i-m}	propagate
1	0	-	impossible
1	1	1	generate

$$c_i = c_i^0 \vee c_{i-m} c_i^1. \quad (5.63)$$

To avoid the slow propagation using the general routing, the idea is to use an RCA, in which each FA computes the sum

$$s_i = c_i^0 \oplus c_i^1 \oplus c_{i-m} \quad (5.64)$$

$$= \begin{cases} c_{i-m} & \text{kill or generate case} \\ \overline{c_{i-m}} & \text{propagate case} \end{cases}. \quad (5.65)$$

Hence, the term c_{i-m} in (5.63) can now be replaced by $\overline{s_i}$ as obtained in the CCC block

$$c_i = c_i^0 \vee \overline{s_i} c_i^1. \quad (5.66)$$

The remaining logic is computed in the CR block (see Fig. 5.30b). Note that the logic of the CR block can be implemented together with the MUX in a single LUT5 per output bit.

The complete architecture is also cheap to pipeline as most of the FFs are otherwise not used in the BLEs that implement the logic. The results presented in [NPP11] clearly indicate that this architecture consumes similar resources compared to the pipelined adder discussed in Sect. 5.5.1 when targeting the same speed, but provides a single cycle latency for large adders beyond 256 bits.

This core idea consisting of a carry-select stage with selection units utilizing the prefix computations in the fast carry chain has been further improved in [LPB19]. Here, several variants of computing the partial additions in the first stage, the prefix network (CCC block in Fig. 5.30), as well as the selection/carry recovery in the last stage have been systematically analyzed. Results were presented for different prefix networks, including Brent-Kung, Han-Carlson, Kogge-Stone, and Sklansky for huge adders computing 1024 to 8192 bits. These show a “quasi linear relationship between prefix tree complexity and performance” [LPB19]. Hence, the data presented in [LPB19] should help to select the right architecture that fits best a given application.

References

- [AB95] E. Abu-Shama and M. A. Bayoumi. "A New Cell for Low Power Adders". In: *International Midwest Symposium on Circuits and Systems*. 1995, pp. 1014–1017. (cit. on p. [107](#)).
- [Alt13] *Stratix V Device Handbook*. Altera Corporation. 2013. (cit. on p. [132](#)).
- [AP02] Massimo Alioto and Gaetano Palumbo. "Analysis and Comparison on Full Adder Block in Submicron Technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 10.6 (2002), pp. 806–823. (cit. on p. [114](#)).
- [Bae+09] Gregg Baeckler, Martin Langhammer, James Schleicher, and Richard Yuan. "Logic Cell Supporting Addition of Three Binary Words". U.S. pat. 7565388. Altera Corporation. 2009. (cit. on p. [132](#)).
- [Bau98] Friedrich L. Bauer. "Zuse, Aiken und der einschrittige Übertrag". In: *Informatik-Spektrum* 21.5 (1998), pp. 279–281. (cit. on p. [114](#)).
- [Bha+15] Partha Bhattacharyya, Bijoy Kundu, Sovan Ghosh, Vinay Kumar, and Anup Dandapat. "Performance Analysis of a Low-Power High-Speed Hybrid 1-bit Full Adder Circuit". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23.10 (2015), pp. 2001–2008. (cit. on pp. [107, 114](#)).
- [BK1] R.P. Brent and H.T. Kung. "A Regular Layout for Parallel Adders". In: *IEEE Transactions on Computers* C-31.3 (1), pp. 260–264. (cit. on p. [122](#)).
- [BL01] Andrew Beaumont-Smith and Cheng-Chew Lim. "Parallel Prefix Adder Design". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2001, pp. 218–225. (cit. on p. [122](#)).
- [BWJ03] Hung Tien Bui, Yuke Wang, and Yingtao Jiang. "Design and Analysis of Low-Power 10-Transistor Full Adders Using Novel XORXNOR Gates". In: *IEEE Transactions On Circuits And Systems II: Analog And Digital Signal Processing* 49.1 (2003). (cit. on p. [107](#)).
- [DNP10] Florent de Dinechin, Hong Diep Nguyen, and Bogdan Pasca. "Pipelined FPGA Adders". In: *International Conference on Field Programmable Logic and Application (FPL)*. IEEE, 2010, pp. 422–427. (cit. on pp. [135, 137](#)).
- [EL04] Miloš D. Ercegovac and Tomás Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004. (cit. on pp. [116, 118, 121, 131](#)).
- [FO01] Michael J. Flynn and Stuart F. Oberman. *Advanced Computer Arithmetic Design*. Wiley-Interscience, 2001. (cit. on p. [111](#)).
- [GKP94] Ronald L. Graham, Donald Ervin Knuth, and Oren Patashnik. *Concrete Mathematics – A Foundation for Computer Science*. Addison-Wesley Professional, 1994. (cit. on p. [115](#)).

- [Har03] David Harris. "A Taxonomy of Parallel Prefix Networks". In: *Asilomar Conference on Signals, Circuits and Systems*. IEEE, 2003, pp. 2213–2217. (cit. on p. 122).
- [Int17] *Intel Stratix 10 Logic Array Blocks and Adaptive Logic Modules User Guide*. Intel Corporation. 2017. (cit. on p. 132).
- [Int18a] *Intel Stratix 10 High-Performance Design Handbook*. Intel Corporation. 2018. (cit. on p. 132).
- [KG16] Petter Källström and Oscar Gustafsson. "Fast and Area Efficient Adder for Wide Data in Recent Xilinx FPGAs". In: *International Conference on Field Programmable Logic and Application (FPL)*. IEEE, 2016. (cit. on p. 135).
- [KKZ16] Martin Kumm, Marco Kleinlein, and Peter Zipf. "Efficient Sum of Absolute Difference Computation on FPGAs". In: *International Conference on Field Programmable Logic and Application (FPL)*. IEEE, 2016, pp. 1–4. (cit. on pp. 133, 134).
- [KS73] Peter M. Kogge and Harold S. Stone. "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations". In: *IEEE Transactions on Computers C-22.8* (1973), pp. 786–793. (cit. on p. 121).
- [Kum+13] Martin Kumm, Martin Hardieck, Jens Willkomm, Peter Zipf, and Uwe Meyer-Baese. "Multiple Constant Multiplication with Ternary Adders". In: *International Conference on Field Programmable Logic and Application (FPL)*. 2013, pp. 1–8. (cit. on p. 132).
- [LPB19] Martin Langhammer, Bogdan Pasca, and Gregg Baeckler. "High Precision, High Performance FPGA Adders". In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 298–306. (cit. on pp. 135, 137, 139).
- [NPP11] Hong Diep Nguyen, Bogdan Pasca, and Thomas B. Preußer. "FPGA-Specific Arithmetic Optimizations of Short-Latency Adders". In: *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2011, pp. 232–237. (cit. on pp. 135, 137, 139).
- [Oka+00] Souichi Okada, Naoya Torii, Kouichi Itoh, and Masahiko Takenaka. "Implementation of Elliptic Curve Cryptographic Co-processor over $GF(2^m)$ on an FPGA". In: *Cryptographic Hardware and Embedded Systems*. Springer, 2000, pp. 25–40. (cit. on p. 104).
- [Pat+07] Dinesh Patil, Omid Azizi, Mark Horowitz, Ron Ho, and Rajesh Ananthraman. "Robust Energy-Efficient Adder Topologies". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2007, pp. 29–37. (cit. on p. 122).
- [RHG14] Marcin Rogawski, Ekawat Homsirikamol, and Kris Gaj. "A Novel Modular Adder for One Thousand Bits and More Using Fast Carry Chains of Modern FPGAs". In: *International Con-*

- ference on Field Programmable Logic and Application (FPL). IEEE, 2014, pp. 1–8. (cit. on pp. 135, 137).
- [Roj97] Raúl Rojas. “Konrad Zuse’s Legacy: The Architecture of the Z1 and Z3”. In: *IEEE Annals of the History of Computing* 19.2 (1997). (cit. on p. 114).
- [Roy+21] Rajarshi Roy, Jonathan Raiman, Neel Kant, Ilyas Elkin, Robert Kirby, Michael Siu, Stuart Oberman, Saad Godil, and Bryan Catanzaro. “PrefixRL: Optimization of Parallel Prefix Circuits using Deep Reinforcement Learning”. In: *Design Automation Conference (DAC)*. ACM/IEEE, 2021, pp. 853–858. (cit. on p. 123).
- [SB00] Ahmed M. Shams and Magdy A. Bayoumi. “A Novel High-Performance CMOS 1-Bit Full-Adder Cell”. In: *IEEE Transactions On Circuits And Systems II: Analog And Digital Signal Processing* 47.5 (2000). (cit. on pp. 107, 114).
- [SE01] P.-M. Seidel and G. Even. “On the Design of Fast IEEE Floating-Point Adders”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2001, pp. 184–194. (cit. on p. 124).
- [Skl60] J. Sklansky. “Conditional-Sum Addition Logic”. In: *IEEE Transactions on Electronic Computers EC-9.2* (1960), pp. 226–231. (cit. on p. 119).
- [Soh+22] Jongwook Sohn, David K. Dean, Eric Quintana, and Wing Shek Wong. “Enhanced Floating-Point Adder with Full Denormal Support”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2022. (cit. on pp. 124, 125).
- [SP06] James M. Simkins and Brian D. Philofsky. “Structures and Methods for Implementing Ternary Adders/Subtractors in Programmable Logic Devices”. U.S. pat. 7274211. Xilinx Inc. 2006. (cit. on p. 132).
- [TVC17] V. Torres, J. Valls, and M.J. Canet. “Optimised CORDIC-based atan2 computation for FPGA implementations”. In: *Electronics Letters* 53.19 (2017), pp. 1296–1298. (cit. on p. 129).
- [Wir04] Michael J. Wirthlin. “Constant Coefficient Multiplication Using Look-Up Tables”. In: *Journal of VLSI Signal Processing* 36.1 (2004), pp. 7–15. (cit. on p. 129).
- [Xil14] *Virtex-6 FPGA Data Sheet: DC and Switching Characteristics (DS152)*. Xilinx. 2014. (cit. on pp. 126, 127).
- [XY98] Shenzhen Xing and W W H Yu. “FPGA Adders: Performance Evaluation and Optimal Design”. In: *IEEE Design & Test of Computers* 15.1 (1998), pp. 24–29. (cit. on p. 135).
- [YN08] Romana Yousuf and Najeeb-ud-din. “Synthesis of carry select adder in 65 nm FPGA”. In: *IEEE Region 10 Conference (TENCON)*. 2008, pp. 1–6. (cit. on p. 135).
- [Zim97] Reto Zimmermann. “Binary Adder Architectures for Cell-Based VLSI and their Synthesis”. PhD thesis. Swiss Federal In-

- stitute of Technology, Zurich, 1997. (cit. on pp. [120](#), [121](#), [122](#), [123](#)).
- [Zus84] Konrad Zuse. *Der Computer – Mein Lebenswerk*. Springer, 1984. (cit. on p. [114](#)).
- [ZW92] N. Zhuang and H. Wu. “A new design of the CMOS full adder”. In: *IEEE Journal on Solid-State Circuits* 27 (1992), pp. 840–844. (cit. on p. [107](#)).



6

CHAPTER 6

Fixed-Point Comparison

All animals are equal, but some animals are more equal than others.

G. Orwell, *Animal Farm*

This chapter studies the comparison of two binary numbers in order to rank them. In a generic-purpose processor, such a comparison is usually performed by subtraction, using an existing integer adder. Assuming that the synthesis tools will remove all the hardware that computes the sum bits, this approach is valid in an application-specific architecture. However, devising integer comparison from first principles allows for specific optimizations in some contexts.

6.1 Introduction

This chapter addresses the construction of the operator whose interface is shown in Fig. 6.1. It compares two w -bits unsigned binary integers X and Y and outputs up to three mutually exclusive Boolean signals $X < Y$, $X = Y$, and $X > Y$.

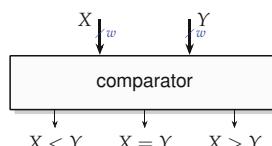


Fig. 6.1 Interface to an integer comparator (it may also output a subset of the comparison bits).

This operator is needed for the floating-point comparator of Sect. 11.5, but also inside many other operators, for instance, to compare exponents in floating-point addition and other summations. Its specialization to a constant Y is used in several places in Chap. 15.

6.1.1 Fixed-Point Considerations

The title of this chapter mentions fixed-point comparison, and Fig. 6.1 only shows a comparator of unsigned integers. Indeed, fixed-point comparison is trivially (i. e., at no cost) reduced to unsigned integer comparison:

- Two unsigned fixed-point numbers X and Y in the same format $\text{ufix}(m, \ell)$ compare the same as the integers $2^\ell X$ and $2^\ell Y$. In other words, the position of the point is irrelevant in the comparison.
- Two unsigned fixed-point numbers X and Y of different formats $\text{ufix}(m_X, \ell_X)$ and $\text{ufix}(m_Y, \ell_Y)$ can be compared by first converting them to the smallest larger common format $\text{ufix}(m, \ell)$ where $m = \max(m_X, m_Y)$ and $\ell = \min(\ell_X, \ell_Y)$ (see Sect. 2.2.3). This conversion does not change the value of either number and therefore does not change the comparison result. Since it involves padding one number with constant zeroes, it enables some of the optimizations related to comparison to a constant operand that will be studied in Sect. 6.3.3.
- In two's complement data, the most significant bit (MSB) has the same weight as it would have in an unsigned representation, but with a negative sign. A consequence is that two signed numbers X and Y of the same format compare the same as the unsigned numbers obtained by exchanging the MSBs of X and Y . Of course this bit exchange is at no cost.

Therefore, the remainder of this chapter focuses on unsigned integer comparison.

6.1.2 Area and Delay Considerations

In principle $X = Y$ can be computed as the logical AND of the individual equality bits $x_i = y_i$. It costs w NXOR gates (which can operate in parallel) and a tree of AND gates that can operate in $\log w$ time.

The Boolean $X < Y$ is the sign bit of the subtraction $X - Y$ and is also its carry out. It can therefore also be computed in logarithmic time (using only the leftmost tree of any of the prefix trees of Chap. 5). Its negation provides $X \geq Y$, and of course $X > Y$ and $X \leq Y$ can be obtained by simply swapping X and Y .

The technique described below does not fundamentally improve these complexities. However, the focus on the comparison operation makes it easier to optimize a comparator for a given context, especially when targeting FPGA technology.

6.2 Basic Binary Tree Comparison

The core idea is simple: given $k = \lceil w/2 \rceil$, X is split into its higher bits X_H and lower bits X_L , or $X = 2^k X_H + X_L$. Similarly Y is split as $Y = 2^k Y_H + Y_L$. Then we have two cases:

- If $X_H = Y_H$ then the order of X and Y is the order of X_L and Y_L .
- If $X_H < Y_H$ or $X_H > Y_H$, we do not need to consider the comparison of X_L and Y_L to decide the ordering of X and Y .

Table 6.1 summarizes these situations. It is then possible to recursively split X_H , Y_H , X_L , and Y_L . This leads to a binary tree. At the leaves are individual bit comparisons.

It remains to encode the three cases of Table 6.1 in binary, which requires two bits. A natural idea is to choose the encoding that we have at the leaves, so that no logic is needed at the leaves: $X > Y$ is encoded by 10, $X < Y$ is encoded by 01, and $X = Y$ is encoded by either 00 or 11.

The previous tree provides the three comparison output bits encoded as two bits. If only $X = Y$ is needed, then the problem obviously becomes simpler, as the equality can be encoded on one bit only (and we obtain the AND tree of XNORs already evoked). However, the problem is not simpler if only $X < Y$ is needed: with the exception of the root node, all nodes in the tree must still distinguish between the three cases $X < Y$, $X_H = Y_H$, and $X_H > Y_H$, which require two bits to encode.

Table 6.1 Divide-and-conquer comparison when X and Y are split in two.

$X \text{ cmp } Y$	$X_H < Y_H$	$X_H = Y_H$	$X_H > Y_H$
$X_L < Y_L$	$X < Y$	$X < Y$	$X > Y$
$X_L = Y_L$	$X < Y$	$X = Y$	$X > Y$
$X_L > Y_L$	$X < Y$	$X > Y$	$X > Y$

6.3 FPGA-Specific Implementations

6.3.1 Exploiting Fast Carry Logic

It is trivial that the $X < Y$ bit is the carry-out of $X - Y$ and can therefore be computed in w LUTs through the fast carry chain. On most modern FPGAs, however, it can be further improved to $w/2$ LUT4s [PZC10] by exploiting the observations made in Sect. 5.4, p. 125. As shown there, most basic logic elements (BLEs) of modern FPGAs allow to compute the full addition of two functions, i. e., $\text{FA}(p(a, b, c, d), q(a, b, c, d), c_i)$. The idea is then to have (a, b, c, d) consisting of two aligned 2-bit chunk X_i and Y_i , respectively, of X and of Y . Then p and q , respectively, compute one bit x'_i and one bit y'_i that compare the same as X_i and Y_i (see Table 6.2). Finally, a subtracter based on the fast carry logic computes $X'_i - Y'_i$, whose sign is that of $X - Y$.

This is the most area-efficient solution if only the $X < Y$ bit is required. It is also the default method used by synthesis tools, which thus implement $X < Y$ on w -bit integers using $\lceil w/2 \rceil$ LUTs.

Similarly, the $X = Y$ bit can be computed as a wide AND of equality comparisons on large chunks. On recent AMD devices, it is possible to chain LUT6 that each compare 3-bit chunks, for a total cost of $\lceil w/3 \rceil$ LUTs.

These two approaches are the most area-efficient, and with fast carry logic, they also provide the best possible latency for sizes up to 64 bits.

Table 6.2 Encoding the comparison of two integers X_i and Y_i as two bits that compare similarly.

	$X_i < Y_i$	$X_i = Y_i$	$X_i > Y_i$
x'_i	0	0	1
y'_i	1	0	0

6.3.2 Two-Level Tree of Fast Carry Logic

For comparisons of very large numbers, the latency of the previous approach (a $w/2$ -bit carry propagation) may be too large. In this case, a good option is to split X and Y in chunks of k bits and perform in parallel the comparison of each chunk. The result of this comparison is then encoded as two k' -bit synthetic integers X' and Y' (see Table 6.2), and these two integers are compared again using a fast carry comparison. In terms of delay, the optimal choice of k and k' is to use $k \approx k' \approx \sqrt{w}$. In a pipeline design, it is also possible to choose k or k' in order to balance the pipeline stages. This is

the choice made by FloPoCo when a comparator is part of a larger pipelined design: k is chosen in such a way that the first level completes the remaining delay available in the current pipeline stage.

This solution is fast, but for a single comparison (e.g., $X < Y$), its cost is much higher than the $w/2$ LUTs of the plain fast carry solution: the area overhead of the second level can be kept small (about $\sqrt{w}/2$), but the first level also adds the cost of the computation of the chunk equality bits (about $w/3$ LUTs).

Other solutions are possible, for instance, a LUT6 allows for a ternary comparison tree instead of the binary tree of Sect. 6.2, while fast carry logic allows for a p -level tree of $\sqrt[p]{w}$ -bit comparators. These solutions can be mixed and matched, but for practical sizes, the plain fast carry approach is always the most area-efficient, while a two-level tree provides very low latency for sizes of several thousand bits.

Hands on: Large comparators in FloPoCo

The following command implements a two-level 1000 bit comparator, each level using fast carry logic:

```
flopoco IntComparator w=1000
```

6.3.3 Comparing to a Constant

In a tree-based approach, the logic optimizer of a synthesis tool will remove parts of the tree if Y is a constant. The main benefit is in LUT-based optimizations: each LUT- α may now compare α bits of the input with the constant. Then, the fast carry logic may be used as previously.

At the time of writing this book, AMD tools do not implement this optimization: they use the logic optimizer for small w and default to a standard comparator for larger w .

Hands on: Large constant comparators in FloPoCo

The following command implements a constant comparator that decomposes the 64-bit input in 13 chunks of 5 bits and links the comparison of these chunks using fast carry logic:

```
flopoco IntConstantComparator flags=1 w=64 \
c=17979737894628297144
```

References

- [PZC10] Stefania Perri, Paolo Zicari, and Pasquale Corsonello. "Efficient Absolute Difference Circuits in Virtex-5 FPGAs". In: *5th IEEE Mediterranean Electrotechnical Conference (Melecon)*. IEEE. 2010, pp. 309–313. (cit. on p. [148](#)).



CHAPTER 7

Sums of Weighted Bits

*I don't know what technology will look like 20 years from now,
but I know one thing: people will still need to compute
additions and multiplications.*

Jean-Michel Muller

A large class of composite fixed-point computations can be expressed as a global sum of weighted bits. This point of view has many advantages, the main one being that the associativity of the sum can be exploited at the bit level to build efficient architectures, called compressor trees, that perform such computations. This chapter defines a data structure, the bit heap, that captures sums of weighted bits. It then discusses the construction of application-specific bit heaps. Finally, it studies the implementation of the corresponding compressor trees.

The summation of several (potentially shifted) fixed-point binary values happens in many fundamental arithmetic blocks. The most prominent example is multiplication, where several partial products have to be added. Specialized multiplications, such as constant multiplication, or squaring, can also be viewed as sums of shifted values. Beyond these, there are many coarser computing blocks that involve sums of product-based terms and therefore can also ultimately be expressed as sums of shifted values. They include, among others, complex products, polynomials used in function evaluation, digital filters and linear transforms used in signal processing, and sums of weighted activations in neural network architectures. As each binary number is itself a sum of weighted bits, any sum of shifted values can be expressed at the bit level as a sum of weighted bits, represented as a *bit heap*. The first main idea in this chapter is that bit heaps capture a very large class of application-specific arithmetic operations.

Of course, summation may be realized by using trees of adders like the ones discussed in Chap. 5. But considering a sum of product-based terms as a global sum of weighted bits enables bit-level optimizations that lead to smaller and/or faster circuits called *compressor trees*. This is an old idea called *carry-save arithmetic* (see Sect. 5.2.6). It was later generalized to include other arithmetic in the context of *merged arithmetic* [Swa80; FS97].

This chapter revisits these concepts in the context of application-specific arithmetic. It introduces *bit heaps* as a useful intermediate representation that helps to design such compressor trees. On the one hand, a bit heap is a generic data structure that captures a very large class of computations in an abstract, bit-level, inherently parallel way. It is at the same time a very high-level mathematical representation that enables algebraic simplifications, and a very low-level one since it describes a computation at the bit level. On the other hand, the bit heap point of view allows for highly efficient circuits in the form of compressor trees. It also simplifies the circuit generation process by factoring out target-specific optimizations, so that they may benefit the whole class of computations that can be expressed as bit heaps.

Section 7.1 introduces the main notions behind bit heaps (weighted bits, compressors, compressor trees) and shows how pervasive and useful this concept can be in application-specific arithmetic. Section 7.2 is about expressing a computation as a bit heap. It shows how to solve a few practical issues such as the efficient management of signed numbers. It also discusses algebraic optimizations that can be performed on a bit heap representation and, last but not least, the construction of truncated bit heaps that compute just right. Section 7.3 discusses the state of the art in compressor tree synthesis. It reviews the elementary compressor families relevant to ASICs and to FPGAs and describes in detail compressor tree generation algorithms that can exploit them.

Most ideas described in this chapter are exploited in the FloPoCo core generator. The reader interested with experimenting with this framework will find the relevant details in Appendix A.3.4, the appendix dedicated to FloPoCo. Section 7.4 provides an experimental evaluation of compressor tree synthesis using this framework.

7.1 Definitions and Motivation

Binary arithmetic represents an integer or fixed-point number as a *sum of weighted bits*:

$$X = \sum_{i=0}^{w_X-1} 2^i x_i. \quad (7.1)$$

We call *weight* a power of two such as the 2^i in the above equation. The integer i is called the *bit position*. There is a bijection between positions and

weights; therefore we will equivalently use the phrases “ x_i is the bit at position i ” and “ x_i is the bit of weight 2^i .”

A general rule in this book is that the index given to a bit corresponds to its binary position, even if it means that the index is negative.

7.1.1 A Multiplier Using a Compressor Tree

Let us introduce the topic with the prevalent operation that involves a weighted sum: multiplication. The product of X by Y , two binary integers, can be expressed as

$$X \times Y = \left(\sum_{i=0}^{w_X-1} 2^i x_i \right) \times \left(\sum_{j=0}^{w_Y-1} 2^j y_j \right) \quad (7.2)$$

$$= \sum_{i=0}^{w_X-1} \sum_{j=0}^{w_Y-1} 2^{i+j} x_i y_j. \quad (7.3)$$

Here, each term $x_i y_j$ is itself a single weighted bit, obtained by the logical AND of x_i and y_j . Therefore, (7.3) is a double sum of weighted bits, and a multiplier is an architecture that computes this double sum. Computing all the $x_i y_j$ bits is easy (a simple AND gate per bit) and embarrassingly parallel. Thus, the main design issue is to compute the value of the sum of all these weighted bits.

This summation can also be represented as a *dot diagram* as shown in Fig. 7.1. There, the horizontal axis represents the bit positions (most significant bits left as usual), and each dot represents one weighted bit participating to the final sum.

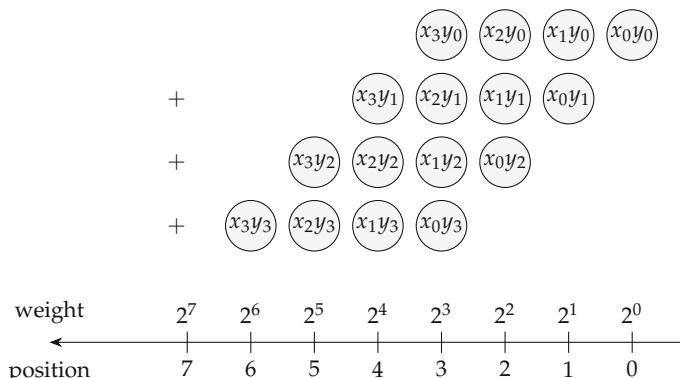


Fig. 7.1 Detailed dot diagram for the product of two 4-bit binary numbers.

One way of performing the multiplication is to use a series of adders, as we do in the paper-and-pencil algorithm. The delay of this solution would be linear in the input size. To reduce this delay, another solution is to build a tree of adders. However, due to the shape of the dot diagram, the adders will be of different sizes, and some of them will input zeroes, which does not seem optimal. Besides, the delay will still be large due to carry propagation, unless fast adders are used as per Sects. 5.3 or 5.5. But then the area will be very large.

An important observation is that all the bits with the same weight can be interchanged in the computation: the vertical order in Fig. 7.1 does not matter. For instance, the same computation can just as well be represented as the dot diagram shown in Fig. 7.2. Instead of adding four 4-bit integers as in Fig. 7.1, the same value can be computed as a sum of one 1-bit value, one 3-bit value, one 5-bit value, and one 7-bit value. This dot diagram view therefore provides us some freedom in the organization of the computation. Compressor trees are architectures that exploit this freedom.

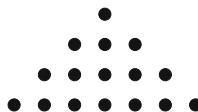


Fig. 7.2 Simplified dot diagram of the product in Fig. 7.1.

The trick here is to completely forget about the *numbers* represented in the dot diagram. If we focus on adding weighted *bits*, these bits do not have to be added row by row: they can be added column by column, or a mixture of both, whatever will give the best architecture. Also, the intermediate carries need not be resolved, but can be saved to be processed later in the architecture, following the concept of *carry-save arithmetic* [EL04] (see Sect. 5.2.6).

Specialized bit-level adders are called *compressors* in the following. The simplest compressor is the full adder (FA) introduced in Sect. 5.2.1. It inputs

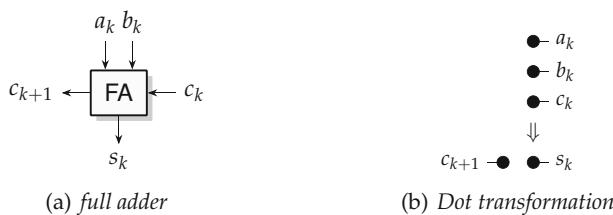


Fig. 7.3 Dot diagram transformation using a full adder.

three bits a_k , b_k , and c_k of the same weight 2^k and rewrites their sum as a 2-bit number: a sum bit s_k of weight 2^k and a carry bit c_{k+1} of weight 2^{k+1} (note how we systematically attempt to use the bit positions as indices). This operation can be interpreted as a transformation of the dot diagram that replaces three bits by two as shown in Fig. 7.3: it *compresses* the dot diagram.

Applying full adders to the dot diagram of Fig. 7.2 results in a smaller dot diagram as illustrated in Fig. 7.4. The idea in compressor tree synthesis is to apply such compression until obtaining a dot diagram with a maximum column height of two. These two rows can then be further compressed by a common adder to get the final result. Indeed, the common integer adder studied in Chap. 5 (with two inputs, a carry in, and a carry out) can also be viewed as a compressor of dot diagrams, as illustrated in Fig. 7.5.

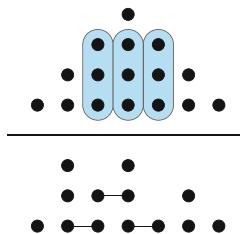


Fig. 7.4 Dot diagrams of the first stage of compression. Sum and carry outputs of full adders are connected by a straight line.

Many more examples of compressors will be reviewed in Sect. 7.3.2, but with the full adder and the row adder, we already have all the components to design a compressor tree. Following the simple greedy strategy to place a full adder whenever there are three or more bits in a column, the initial bit array can be compressed in four stages as shown in Fig. 7.6. The corresponding compressor tree circuit is shown in Fig. 7.7. Note that the four FAs having one input set to zero in Fig. 7.7 could be replaced by the less expensive half adders (HAs).

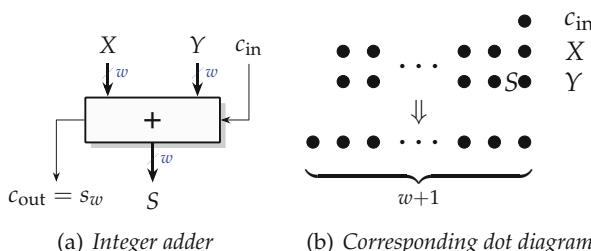


Fig. 7.5 The integer adder viewed as a compressor.

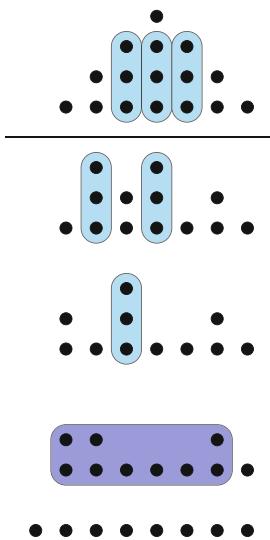


Fig. 7.6 Successive dot diagrams of a complete compression.

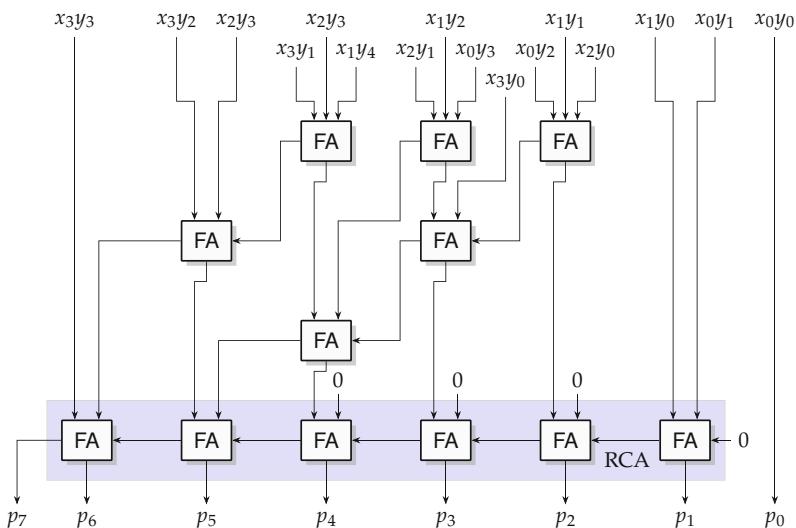


Fig. 7.7 Compressor tree of Fig. 7.6.

7.1.2 From Bit Arrays to Bit Heaps

In the multiplier literature, the dot diagram is usually called a *bit array*, and indeed Fig. 7.1 represents an array. In this book, we call them *bit heaps*, because the shape of the dot diagram can be much less regular than what the word “array” suggests.

Indeed, bit heaps are not limited to computing just multiplication. For instance, computing a multiply-accumulate (MAC) $XY + A$ adds only one line to the bit heap of the product, as illustrated by Fig. 7.8.

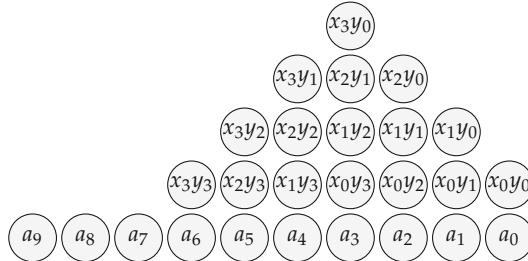


Fig. 7.8 Bit heap computing $XY + A$.

When computing the value of this bit heap, the additional line will entail at most one additional level of compression. Compared to multiplication alone, the area overhead is at most linear in the addition size n , and the delay overhead is at most one compressor delay. This is to be compared to a separate adder behind the multiplier, which would either add n to both delay and area or (using a fast adder) $\log n$ to the delay and $n \log n$ to the area. We observe that such merged arithmetic (merging several operations into a single bit heap) may lead to better area and latency [Swa80; FS97]. Again, this is due to an optimization process that is (1) global, forgetting where bits come from, and (2) at the bit level rather than at the word level.

Such merged arithmetic has been shown to lead to more efficient bit-level implementations of inner products [Swa80; FS97], of the arithmetic parts of data-flow graphs [VBI08], of elementary functions [SDP11; DIS13; Che15; De +17], and of various signal processing kernels [Par+11; BG09; Vol+19]. It is therefore a core ingredient in application-specific arithmetic.

A natural question is: What is the class of operations that can be merged in a bit heap?

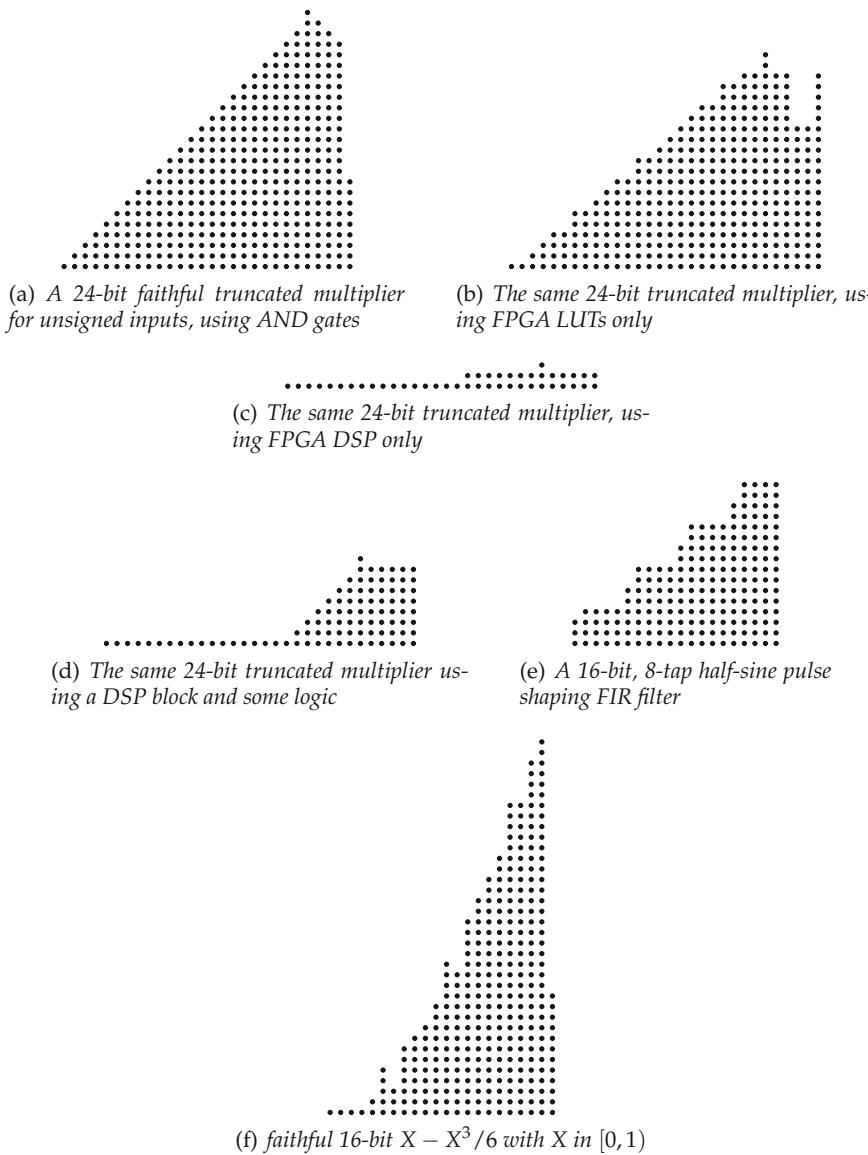


Fig. 7.9 Bit heaps obtained in FloPoCo have various shapes and sizes.

Obviously, the sum of two bit heaps is itself a bit heap. Similarly, the product of two bit heaps is itself a bit heap: it can be developed as a single big sum. By induction, any multivariate polynomial with fixed-point inputs may be expressed as a single sum of weighted bits. This includes addition and multiplication on complex numbers, sums of products and sums of squares (for linear algebra operators or signal processing transforms), polynomials used to approximate elementary functions, digital filters, etc. For instance, Sect. 7.2.3 will detail the construction of a bit heap for the polynomial $X - X^3/6$.

Besides, the bits may come from table lookup or other arbitrary components, which further enlarges the class of functions where a bit heap is relevant. Many examples will be given in this book. Figure 7.9 shows some bit heaps obtained for various components in FloPoCo.

It should also be noted that an operator may involve several bit heaps. For illustration, Fig. 7.10 shows the various bit heaps involved in an architecture [DIS13] that computes both sine and cosine in fixed point. The construction

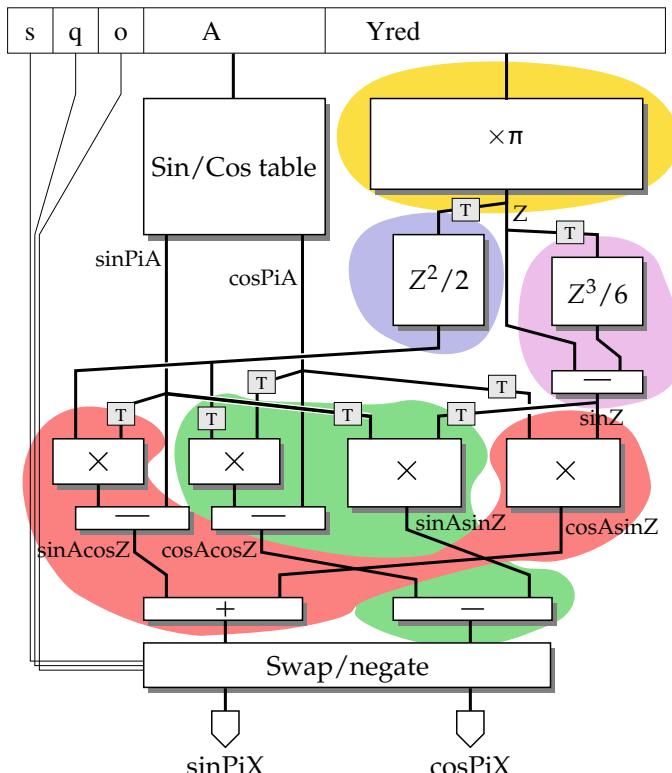


Fig. 7.10 Various bit heaps in a sine/cosine architecture.

of the architecture itself will be detailed in Chap. 20; here we just want to point out that it is not uncommon to have several bit heaps in one operator.

Conversely, in a figure such as Fig. 7.10, an adder or subtractor does not necessarily translate to a component in the architecture: it can be realized as part of a bit heap, merged with other operations.

Another thing we can point out from this example is that some of the bit heaps involved in Fig. 7.10 could be merged into fewer but larger ones. This could improve speed, but it will also prevent reusing intermediate results: it may therefore be counterproductive in terms of resources. This is best understood on an example. In Fig. 7.10, the output Z of the $\boxed{\times\pi}$ operator is input to several operators, e.g., $\boxed{Z^2/2}$ and $\boxed{Z^3/6}$, among others. It is possible to merge the $\boxed{\times\pi}$ and the $\boxed{Z^2/2}$ operator into a single bit heap computing $\boxed{\frac{\pi^2 Y_{\text{red}}^2}{2}}$ and similarly merge the $\boxed{\times\pi}$ and the $\boxed{Z^3/6}$ operator in another bit heap. But then, the value of $Z = \pi Y_{\text{red}}$ is no longer shared and is effectively recomputed twice, which will likely consume more resources.

Hence, one has to carefully trade in this case between two of the main techniques introduced in Chap. 1: *resource sharing* and *operator fusion*.

7.1.3 Bit Heaps for Portable Application-Specific Arithmetic

As Fig. 7.11 illustrates, expressing complex composite operators in terms of bit heaps can also be viewed as an elegant way to separate:

- optimizations that are of algorithmic or mathematical nature (the construction of the bit heap), and
- optimizations that depend on the target technology and target performance (the construction of an architecture computing the value of a bit heap).

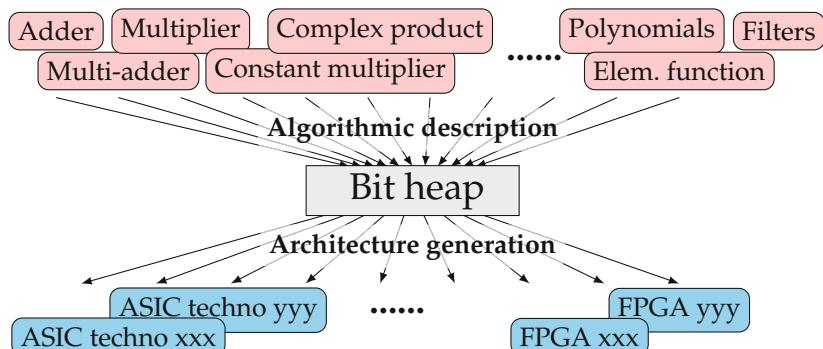


Fig. 7.11 Bit heaps for application-specific arithmetic.

This alone is a strong case for a single, centralized, versatile bit heap framework (one will be presented in Appendix A.3.4). When expressing as much as possible of our operators as bit heaps, it is possible to delegate their implementation to a generator of compressor trees optimized for the target technology.

Each algorithmic description can thus be optimized for many technologies, and each new optimization of the compressor trees brings benefits to many operators. It also makes arithmetic design future-proof. When a new technology appears (a new FPGA, a new process node), it suffices to focus on the compressor tree generator, and all the bit heap-based operators will instantly be optimized for this new technology.

7.2 Expressing a Computation as a Bit Heap

This section first reviews some of the techniques that are needed to make merged arithmetic practical and then some optimizations that may be performed at the algebraic level.

7.2.1 Managing Constant Bits

Some of the bits added to a bit heap are constant bits, for instance:

- half-ulp bits needed to round by truncation (as per Sect. 3.1.3) terms added to the bit heap,
- the half-ulp bit needed to round to the nearest the final result of the bit heap,
- sign extension bits when managing two's complement signed numbers, as explained in Sect. 7.2.2.

The sum of all these constant bits is known in advance: it can be precomputed at the construction of the circuit, such that at most one constant bit remains per column. This is an old trick, widely used in the design of Baugh-Wooley signed multipliers [BW73] and multi-input adders [EL04]. In these cases, the addition of the constant bits was computed by hand. In the context of application-specific arithmetic, it is simpler, safer, and more flexible to have it computed by a bit heap framework, such as the one that will be presented in Appendix A.3.4.

As a side effect, note that the addition of one half-ulp needed to achieve rounding to the nearest can usually be merged in the global compression for free. It is the case for all the bit heaps of Fig. 7.10, for instance. We will point this out when relevant.

7.2.2 Managing Signed Numbers

So far, our detailed examples of bit heaps were unsigned: the partial products were positive integers, and their sum was also a positive integer. Signed numbers are classically represented using two's complement notation [EL04]. It is possible to design bit heaps operating directly on positive and negative bits [JPG06], but this section shows that managing two's complement signed numbers in a bit heap costs very little, as most of the overhead can be compressed into one row of constant bits.

Most numbers added to the bit heap do not extend to the most significant bit (MSB) of the bit heap. When such a number is signed, it must be sign-extended: its MSB (which is the sign bit) must be replicated all the way up to the MSB of the bit heap [EL04].

Take, for example, a 4-bit signed integer number $X = x_3x_2x_1x_0$. When adding this number to a, say, 8-bit number, we have to replicate the sign bit to get $X' = x_3x_3x_3x_3x_3x_2x_1x_0$. The reader may check that this still represents the same integer as X (see also Sect. 2.2.3).

Performing the sign extension this way would add many additional bits to a bit heap, some of them with large fanout.¹ It is not an uncommon situation to add many signed numbers requiring a signed extension. For instance, consider a multiplier of a signed integer X by an unsigned one Y (a slightly simpler case than the more common signed by signed multiplier). It will involve a bit array such as that of Fig. 7.1. However, each row is now signed (since it is zero or a shifted copy of X) and must therefore be sign-extended to the MSB of the result before addition. This sign extension would entail the bit heap shown in Fig. 7.13a.

We can use here a classic trick from two's complement multipliers: to replicate the sign bit s from index p to index q , where q is the MSB of the bit heap:

- we add, instead of s , the complement \bar{s} of s at position p ,
- then we add a string of constant 1s stretching from bit p to bit q .

$$\begin{array}{r}
 & \bar{s} \\
 + & 1 1 1 1 1 1 1 1 \\
 \hline
 = & \bar{s} s s s s s s s \\
 & \quad \uparrow \quad \uparrow \\
 \text{index: } & q \quad p
 \end{array}$$

Fig. 7.12 Dot transformations for efficient sign extension.

¹ The fanout is the number of gate inputs connected to a single output.

This is illustrated in Fig. 7.12. If $s = 1$, the result is a vector of 1s, which is indeed the extension of s . If $s = 0$, the sum is a vector of 0s, except the bit at position $q + 1$ which is 1. However, this 1 bit is ignored by the modulo arithmetic of two's complement with MSB q . In both cases, we have extended s from position p to position q .

For a bit heap adding several signed terms, this technique adds a lot of constant 1s to the bit heap (Fig. 7.13b). However, this is not an issue, as all these constant bits can be compressed into one single constant row, as already discussed in Sect. 7.2.1: the overhead of a bit heap accepting signed numbers, with respect to an unsigned bit heap, is at most one row of constant ones (Fig. 7.13c).

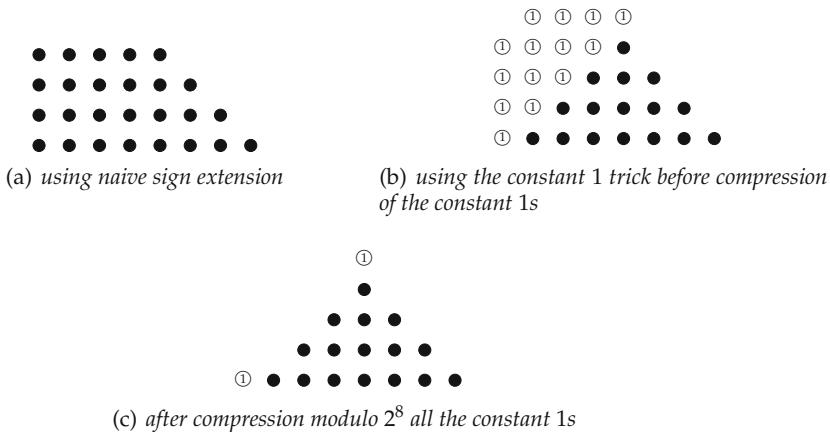


Fig. 7.13 Dot diagrams for a 4-bit signed \times unsigned multiplier.

7.2.2.1 Subtracting a Number

Subtracting a number corresponds to adding the negative of that number. This is simply realized by adding the bitwise complement (with sign extension) and a constant 1 to the LSB position of the subtrahend. Again, this constant one is compressed with other possible constant bits.

7.2.3 Algebraic Optimizations

Some simple optimizations can also be used in bit heaps computing algebraic expressions. The most common are presented in the following.

Let $x \in \{0, 1\}$ being a single bit in a weighted sum expression. Then, the addition of two x with the same weight can be performed by bit shifting, i. e., moving the bit to the next higher column (see Fig. 7.14a). Clearly, adding up more than two x , the rule of Fig. 7.14a can be applied several times, leading to a constant multiplication. Figure 7.14b shows the example for $3x$. As $3 = 11_2$, the three bits can be replaced by the bit pattern xx . This easily extends to other constants as well.

Another common rule is that $x \cdot x = x$, or more general, $x^k = x$. This often appears when computing squares or other powers of an input.

The addition of a constant “1”, i. e., $x + 1$, can be precomputed and leads to \bar{x} for the sum and x for the carry as illustrated in Fig. 7.14c. When the same variable appears several times in the same column as indicated in Fig. 7.14d (with $y \in \{0, 1\}$), a precomputation is also often possible. This can also help to reduce the height of the largest column(s), e. g., to decrease the number of levels in the compressor tree. In the case that a bit and its negation is found in the same column, it can be replaced by a constant “1” as shown in Fig. 7.14e.

Note that all the discussed transformations shown in Fig. 7.14 actually also work the other way around, e. g., the rule described in Fig. 7.14c could be applied this way to replace a non-constant bit with a constant bit which might be merged with other constant bit(s) of the same column.

$$\begin{array}{cccccc}
 & & x & & & \\
 & x & & x & & 1 & & y & & \bar{x} \\
 & x & & x & & x & & xy & & x \\
 \Downarrow & & \Downarrow & & \Downarrow & & \Downarrow & & \Downarrow & \\
 x \cdot & & x \cdot x & & x \cdot \bar{x} & & xy \cdot \bar{x}y & & 1 \\
 \text{(a) } 2x & \text{(b) } 3x & \text{(c) } x + 1 & \text{(d) } y + xy & \text{(e) } x + \bar{x} = 1
 \end{array}$$

Fig. 7.14 Algebraic dot transformations.

An enlightening example is a third-order Taylor formula for the sine: $\sin(X) \approx X - X^3/6$. It was one of the bit heap-based blocks of Fig. 7.10 that will be fully studied in Chap. 20.

This formula can be evaluated using two standard multiplications (to compute X^3), a multiplication by the constant $1/6$, and a subtraction. However, it can also be expressed directly as a single bit heap as follows. Using in X^3 the ufix(m, ℓ) fixed-point representation of X (defined as (2.6) of Chap. 2), we get

$$X^3 = \left(\sum_{i=\ell}^m 2^i x_i \right)^3 \quad (7.4)$$

$$= \sum_{i=\ell}^m 2^{3i} x_i^3 + \sum_{\ell \leq i \neq j \leq m} 3 \cdot 2^{i+2j} x_i x_j^2 + \sum_{\ell \leq i < j < k \leq m} 6 \cdot 2^{i+j+k} x_i x_j x_k . \quad (7.5)$$

Here we may apply the simplifications $x_i^k = x_i$ and $2 \cdot 2^p a = 2^{p+1} a$ (Fig. 7.14a). We get

$$X^3 = \sum_{\ell \leq i \leq m} 2^{3i} x_i + 3 \left(\sum_{\ell \leq i < j \leq m} (2^{i+2j} + 2^{2i+j}) x_i x_j + \sum_{\ell \leq i < j < k \leq m} 2^{i+j+k+1} x_i x_j x_k \right) . \quad (7.6)$$

Ignoring for a moment the multiplication by 3, the three summations in (7.6) contribute, respectively, for a w -bit input, w , $w(w - 1)$, and $w(w - 1)(w - 2)/6$ terms [A000292] to the bit heap. It may sound a lot of bits, and indeed it grows in w^3 . However, let us compare this with the cost of two successive multipliers, the first a $w \times w$ -bit one and the second a $2w \times w$ -bit one since it inputs the exact result of the first, on $2w$ bits. The total number of terms in these two multipliers is $3w^2$, and $3w^2$ is smaller than $w + w(w - 1) + w(w - 1)(w - 2)/6$ only for $w \geq 14$. In other words, for $w < 14$, a single bit heap computing X^3 is more area-efficient than two multipliers. It should also be consistently faster, since it involves a single compression process. This result illustrates the power of our algebraic simplifications. However, we will also soon see that in practice, many of these bits need not be computed at all (and this applies to both approaches, the single bit heap, or the two multipliers).

The multiplication by 3 could be obtained by duplicating bits in the bit heap (Fig. 7.14b). If we were interested in computing exactly X^3 , it would be clever to first compress all the terms to be multiplied by 3 into two lines and then duplicate only these two lines before completing the global compression. However, as we are interested in $X - X^3/6$, these multiplications by 3 disappear, and from (7.6), we obtain

$$X - \frac{X^3}{6} = \sum_{\ell \leq i \leq m} 2^i x_i - \frac{1}{3} \sum_{\ell \leq i \leq m} 2^{3i-1} x_i - \sum_{\ell \leq i \neq j \leq m} 2^{i+2j-1} x_i x_j - \sum_{\ell \leq i < j < k \leq m} 2^{i+j+k} x_i x_j x_k . \quad (7.7)$$

The subtractions can be managed as per Sect. 7.2.2, so we leave them for readability. Now, we have only $m - \ell$ bits of the bit heap to actually divide by 3, although they are spread as a $3w$ -bit number (due to the weights 2^{3i-1}). One option is to use a variation of the divider by 3 that will be later presented in Chap. 13, possibly slightly optimized to account for the fact that the only non-zero bits have positions $3i - 1$. This adds only one row of

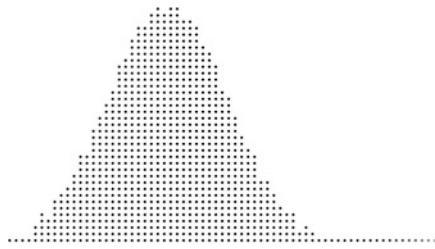
bits to the bit heap, albeit after a delay. This delay is not a problem: these late bits may be scheduled to participate in later compression stages. The bit heap obtained with this option is shown in Fig. 7.15a.

Another option is to compute the division by 3 by expanding the remaining $1/3$ into an infinite sum of weighted bits: $1/3 = 0.010101010\dots = \sum_{j \leq -1} 2^{2j}$. Then (7.7) becomes

$$R^* = \sum_{\ell \leq i \leq m} 2^i x_i - \sum_{\substack{\ell \leq i < j \leq m \\ j \leq -1}} 2^{3i-1-2j} x_i - \sum_{\ell \leq i \neq j \leq m} 2^{i+2j-1} x_i x_j - \sum_{\ell \leq i < j < k \leq m} 2^{i+j+k} x_i x_j x_k. \quad (7.8)$$

The bit heap obtained with this option is shown in Figs. 7.15b, and 7.16 decomposes it into the various sub-sums of (7.8).

The fact that the bit heaps in Fig. 7.15 now extend to infinity will not be a problem in practice: the actual circuit must output a finite number of bits anyway, and even when the bit heap is finite, it is usually too large and must be rounded. Rounding an infinite number of bits is no more difficult. Actually, in a circuit computing just right, it is often possible to ignore a large part of the exact bit heap and still achieve good enough accuracy. Let us now address this problem.



(a) using (7.7), with an external divider by 3



(b) With a multiplication by $1/3$ in the bit heap

Fig. 7.15 Exact bit heaps for $X - X^3/6$ with X in $\text{ufix}(-1, -16)$.

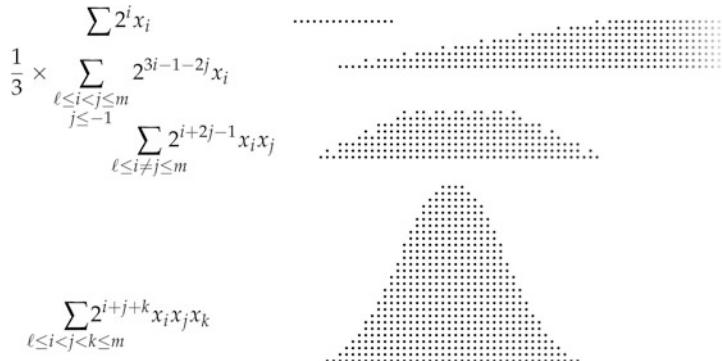


Fig. 7.16 The bit heap of Fig. 7.15b decomposed in its components.

7.2.4 Truncating a Bit Heap for Last-Bit Accuracy

We assume here that the exact bit heap provides more bits than needed in the output. We also assume, as per Sect. 3.1.8, p. 76, that the output format serves as a specification of the accuracy of the operator: the operator should be faithful to its output format or, in other terms, accurate to its last bit. This section shows how to transform the exact bit heap into a smaller one that respects this specification at the minimal cost.

This section is illustrated with two examples: a simple multiplier and the more convolved example of $X - X^3/6$, in both cases with inputs and outputs sharing the same $\text{ufix}(-1, \ell)$ format for some integer $\ell < 0$. However, the techniques presented here apply to any bit heap, including those that have more than one input (e.g., a complex product, a sum of products or a sum of squares). These techniques are also trivial to adapt to signed fixed-point formats or to an output format different from the input format. Indeed, signedness and input format matter little as far as truncation is concerned: in all this section, the parameter ℓ essentially represents the LSB of the output format.

7.2.4.1 Overall Error Analysis

We denote R^* the exact sum that would be computed by the bit heap (for some value of the inputs). This value may be representable exactly, but on more bits than the output format allows (e.g., the exact product of two the $\text{ufix}(-1, \ell)$ is an $\text{ufix}(-1, 2\ell)$ number). This value may also not be finitely representable in binary, as in the $X - X^3/6$ example, or as in the multiplier by π visible in Fig. 7.10 (since π is irrational, the exact product by π of any non-zero input is also irrational).

In any case, our purpose is to build an operator that is faithful to the output format. The output R of the operator is then no longer the exact result R^* , and we may define the error δ_R :

$$\delta_R = R - R^* . \quad (7.9)$$

The faithful rounding constraint (defined in Sect. 3.1.8) is that this error must remain strictly smaller, in absolute value, than the unit in the last place (ulp) of the result

$$\bar{\delta}_R < 2^\ell \quad (7.10)$$

(reminding that $\bar{\delta}_R$ denotes the maximum possible value of $|\delta_R|$ considering all the possible input values). Following the error decomposition of Sect. 3.3, this error is partly due to the fact that we will use a truncated bit heap that computes an approximation \tilde{R} of R^* and partly due to the rounding of \tilde{R} to get R :

$$\begin{aligned} \delta_R &= R - R^* \\ &= R - \tilde{R} + \tilde{R} - R^* \end{aligned} \quad (7.11)$$

$$= \delta_{\text{final round}} + \delta_{\tilde{R}} . \quad (7.12)$$

By triangular inequality, (7.12) yields $|\delta_R| \leq |\delta_{\text{final round}}| + |\delta_{\tilde{R}}|$, which defines a bound $\bar{\delta}_R$ on $|\delta_R|$:

$$\bar{\delta}_R = \bar{\delta}_{\text{final round}} + \bar{\delta}_{\tilde{R}} . \quad (7.13)$$

The final rounding is based on the identity $[x] = [x + 1/2]$: as Fig. 7.17 illustrates, rounding of \tilde{R} to R will be achieved by adding one half-ulp to the bit heap (a constant 1 in position $\ell - 1$), then computing the sum thanks to a compressor tree, and then truncating this sum by discarding all the bits of positions smaller than ℓ . This entails a rounding error bound $\bar{\delta}_{\text{final round}} = 2^{\ell-1}$; therefore the constraint on the accuracy of \tilde{R} together with (7.10) becomes

$$\bar{\delta}_{\tilde{R}} < 2^{\ell-1} . \quad (7.14)$$

The remainder of this section discusses ways to ensure this constraint at the least possible cost.

The core idea is to truncate as many bits as possible from the right of the exact bit heap. We call *exact bit heap* an expression of R^* as a (possibly infinite) sum of weighted bits. For example, an exact bit heap for the ufix($-1, l$) multiplication is depicted as the full lozenge in Fig. 7.17. Its expression is

$$R^* = X \times Y = \sum_{\substack{-1 \geq i \geq \ell \\ -1 \geq j \geq \ell}} 2^{i+j} x_i y_j. \quad (7.15)$$

For $R^* = X - X^3/6$, the two Eqs. (7.7) and (7.8) define two alternative exact bit heaps (both infinite). Note that there also exists many possible exact bit heaps for multiplication—Chap. 8 will introduce some of them.

Now, let us consider what happens when we remove (ignore, discard) bits from an exact bit heap. Removing a bit in position i entails, for at least one input, a contribution of -2^i to the error $\delta_{\tilde{R}} = \tilde{R} - R^*$. We must therefore compute all the bits whose position i is at least $\ell - 1$: ignoring any of them would lead, for at least one input, to an error $2^i \geq 2^{\ell-1}$, hence, a violation of the faithful rounding constraint (7.14). Conversely, we could randomly remove one bit at position $\ell - 2$, and (7.14) would still be satisfied. However, we could also, for the same error, remove two bits at position $\ell - 3$ or 4 bits at position $\ell - 4, \dots$. Assuming that all the bits have the same hardware cost,² we are encouraged to remove first the rightmost bits.

7.2.4.2 Parameters of a Truncated Bit Heap

A consequence of this observation will be that we can define an efficient truncation by only two parameters: an integer ℓ_{ext} that defines the rightmost column of the truncated bit heap and the number t of partial products removed in this rightmost column. For instance, Fig. 7.17 shows a faithful truncated multiplier such that $\ell_{\text{ext}} = \ell - 4$ and $t = 7$ (the truncated bits are represented with empty circles). The key point here is that it does not make much sense to truncate bits in column $\ell_{\text{ext}} + 1$, since for each bit saved there we could save two bits in the column ℓ_{ext} . Similarly, it does not make much sense to compute a bit in column $\ell_{\text{ext}} - 1$ if we truncate bits in column ℓ_{ext} : it will be more accurate for the same price to compute a bit in column ℓ_{ext} instead.

This being established, let us define $\Delta(\ell_{\text{ext}}, t)$ as the weighted sum of all the truncated bits, assuming they are all equal to 1. This value is an upper bound of the error due to truncation,³ so we may use $\bar{\delta}_{\tilde{R}} = \Delta(\ell_{\text{ext}}, t)$ in the constraint (7.14).

In the multiplier example, the maximum error $\Delta(\ell_{\text{ext}}, t)$ is attained when the two inputs are equal to the maximal possible input 0.111..111: it is the weighted sum of all the truncated bits when they are all equal to 1 or

² This simplifying hypothesis is true for the multiplier, but not, for example, when using a divider by 3 to compute some bits in (7.7) and an AND gate to compute some others. In our presentation we will ignore such considerations for simplicity, but they can easily be managed in the method presented here.

³ This bound is attained in the two examples taken in this chapter, but we will show in the sequel why it may be pessimistic in some cases.

(Fig. 7.17)

$$\Delta(\ell_{\text{ext}}, t) = t \cdot 2^{\ell_{\text{ext}}} + 2^{2\ell} \left(1 + 2 \cdot 2^1 + 3 \cdot 2^2 \dots + n \cdot 2^{n-1} \right) \quad (7.16)$$

where to simplify notations $n = \ell_{\text{ext}} - 2\ell$ is the width of the triangle in blue in Fig. 7.17. We may simplify this equation using the following formula [A000337], easily proven by recurrence:

$$\sum_{i=0}^{n-1} (i+1)2^i = (n-1) \cdot 2^n + 1 \quad (7.17)$$

hence,

$$\Delta(\ell_{\text{ext}}, t) = t \cdot 2^{\ell_{\text{ext}}} + ((n-1) \cdot 2^n + 1)2^{2\ell} . \quad (7.18)$$

As a general rule, $\Delta(\ell_{\text{ext}}, t)$ can be decomposed as follows:

$$\Delta(\ell_{\text{ext}}, t) = t \cdot 2^{\ell_{\text{ext}}} + \Delta^{\text{minor}}(\ell_{\text{ext}}) \quad (7.19)$$

where $\Delta^{\text{minor}}(\ell_{\text{ext}})$ represents the sum of all the weighted bits to the right of ℓ_{ext} (in Fig. 7.17, the triangle in blue)

$$\Delta^{\text{minor}}(\ell_{\text{ext}}) = \sum_{i=\ell_{\text{min}}}^{\ell_{\text{ext}}-1} h_i \cdot 2^i \quad (7.20)$$

where ℓ_{min} is the LSB of the bit heap (it can be $-\infty$, but in this case the infinite sum will nevertheless have a finite value) and h_i denotes the height of the column of the bit heap at position i . Notice that $\Delta^{\text{minor}}(\ell_{\text{ext}})$ does not depend on t . Remark also that ℓ_{ext} has an architectural meaning: it defines the least significant bit (LSB) of the bit heap that will actually be compressed.

It can be tempting [DRC14] to look for closed-form formulas such as (7.18) for $\Delta(\ell_{\text{ext}}, t)$, but we did it here only to show the futility of the exercise: on the one hand, (7.18) does not give us a formula that computes the largest values of ℓ_{ext} and t such that constraint (7.14) is met. On the other hand, it is very easy to write a program that does. Such a program essentially attempts to remove bits from the right of the bit heap, one at a time, and evaluates the corresponding $\Delta(\ell_{\text{ext}}, t)$ until (7.14) is no longer satisfied. This programmatic approach has the huge advantage that it works for any bit heap, whatever its shape. Before developing such programs, let us first refine the error model.

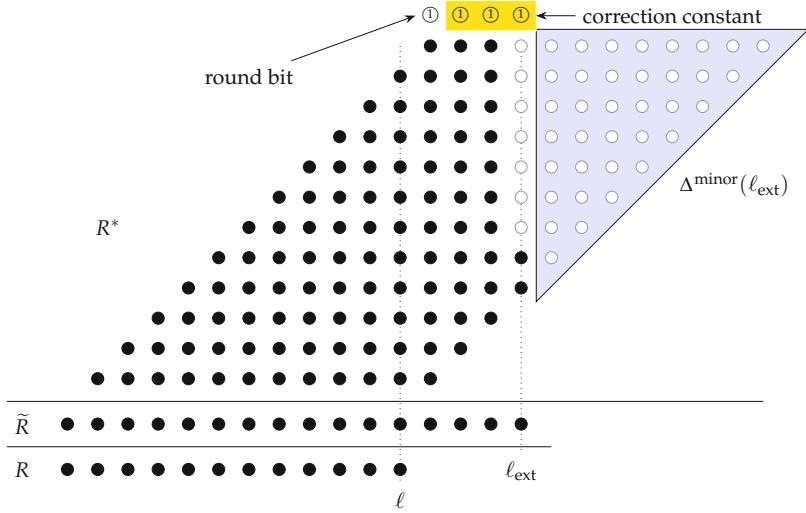


Fig. 7.17 Truncated bit array for a faithful multiplier with inputs and outputs in $\text{ufix}(-1, -12)$ format. The error $\delta_{\tilde{R}}$ is bounded as follows: $\delta_{\tilde{R}} \in [-1793 \cdot 2^{-24}, 1792 \cdot 2^{-24}]$, therefore $\bar{\delta}_{\tilde{R}} < 2048 \cdot 2^{-24} = 2^{-13}$; constraint (7.14) is met.

7.2.4.3 Constant Correction

So far, the truncation error $\delta_{\tilde{R}} = \tilde{R} - R^*$ is always negative, since we only remove bits from the bit heap when truncating: $\delta_{\tilde{R}} \in [-\Delta(\ell_{\text{ext}}, t), 0]$, hence, $\bar{\delta}_{\tilde{R}} = \Delta(\ell_{\text{ext}}, t)$. Thus, it is interesting to recenter $\delta_{\tilde{R}}$ on zero, and this can be done by simply adding a constant to the bit heap [SS93]. If we could add the constant $\frac{\Delta(\ell_{\text{ext}}, t)}{2}$ to the bit heap, the truncation error would become $\delta_{\tilde{R}} \in [-\frac{\Delta(\ell_{\text{ext}}, t)}{2}, \frac{\Delta(\ell_{\text{ext}}, t)}{2}]$, thus halving $\bar{\delta}$. Since the constraint (7.14) on $\bar{\delta}$ does not change, this will allow us to truncate more (doubling $\Delta(\ell_{\text{ext}}, t)$).

In practice, however, the weights of these constant bits can only be between $2^{\ell-2}$ and $2^{\ell_{\text{ext}}}$: adding $2^{\ell-1}$ would automatically violate the constraint (7.14), while adding constant bits to the right of column ℓ_{ext} would mean enlarging the bit heap, which is not wanted. Hence, the constant can only be an approximation of $\frac{\Delta(\ell_{\text{ext}}, t)}{2}$.

A very good heuristic is to systematically add the largest possible correction constant:

$$C = 2^{\ell-1} - 2^{\ell_{\text{ext}}} . \quad (7.21)$$

This is a string of ones from positions $\ell - 2$ to ℓ_{ext} (note that there is also the rounding bit at position $\ell - 1$, see Fig. 7.17).

With this constant, the interval of the error becomes

```

1 // Truncation of a bitheap faithful to  $2^\ell$ 
2 computeTruncation( $\ell$ ) :
3   // initialization
4    $\ell_{\text{ext}} \leftarrow \ell_{\text{min}}$  // start from the right
5    $t \leftarrow 0$  // no bit is removed
6
7   // First loop: determine  $\ell_{\text{ext}}$ 
8   while (  $(t+1) \cdot 2^{\ell_{\text{ext}}+1} + \Delta^{\text{minor}}(\ell_{\text{ext}}+1) < 2^\ell$  )
9      $\ell_{\text{ext}} \leftarrow \ell_{\text{ext}} + 1$ 
10
11  // Second loop: determine  $t$ 
12  while (  $(t+2) \cdot 2^{\ell_{\text{ext}}} + \Delta^{\text{minor}}(\ell_{\text{ext}}) < 2^\ell$  )
13     $t \leftarrow t + 1$ 
14
15  return ( $\ell_{\text{ext}}, t$ )

```

Listing 7.1 Truncation of an exact bit heap.

$$\delta_{\tilde{R}} \in [C - \Delta(\ell_{\text{ext}}, t), C] \quad . \quad (7.22)$$

The upper bound C obviously satisfies (7.14). The absolute value of the (negative) lower bound can be rewritten, using (7.19)

$$\Delta(\ell_{\text{ext}}, t) - C = t \cdot 2^{\ell_{\text{ext}}} + \Delta^{\text{minor}}(\ell_{\text{ext}}) - (2^{\ell-1} - 2^{\ell_{\text{ext}}}) \quad . \quad (7.23)$$

The constraint (7.14) can now be rewritten $\Delta(\ell_{\text{ext}}, t) - C < 2^{\ell-1}$, or

$$(t+1) \cdot 2^{\ell_{\text{ext}}} + \Delta^{\text{minor}}(\ell_{\text{ext}}) < 2^\ell. \quad (7.24)$$

Note that for very small output word sizes in bit arrays containing signed numbers, it may happen that bits from sign extension are covered by $\Delta^{\text{minor}}(\ell_{\text{ext}})$ (the blue triangle in Fig. 7.17). These bits have a negative weight, leading to positive truncation errors and, hence, a lower correction constant C (which might be even zero). In this case, the general form (7.23) has to be checked in Listing 7.1.

7.2.4.4 A Generic Algorithm

Listing 7.1 describes a simple universal algorithm that determines the maximal acceptable values of ℓ_{ext} and t . It consists of two successive loops: a first one increases ℓ_{ext} for $t = 0$ as long as (7.24) is respected, followed by a second one that attempts to increase t as long as (7.24) is respected. The interested reader will find several instances of this algorithm in FloPoCo (sometimes well hidden). It is a good idea, in the implementation of Listing 7.1, to scale all the weighted sums by $2^{-\ell_{\text{min}}}$: then the two loops only manipulate arbitrary-precision integers, which are very simple to use in modern lan-

guages (e.g., using the `mpz_class` for C++, the default integer arithmetic in Python, BigInts in Julia, etc).

All what this algorithm needs is an implementation of $\Delta^{\text{minor}}(p)$, as per (7.20).

We still have the small problem of initializing this algorithm in the case when the bit heap is infinite. Here we simply use, in the computation of $\Delta^{\text{minor}}(\ell_{\text{ext}})$, the convergence of the series $\sum_{i < p} 2^i = 2^p$. For instance, for the bit heap of Fig. 7.15b, truncating each of the 8 infinite lines after position p will contribute 2^p to $\Delta^{\text{minor}}(\ell_{\text{ext}})$. This is illustrated by the eight extra dots in the rightmost column of Fig. 7.18: these dots alone completely capture the infinite tail of Fig. 7.15b. This initial truncation must be performed before using Listing 7.1, at a random position that is far enough to the right for Listing 7.1 to actually enter the first loop.

Note that the granularity of the error added to $\delta_{\tilde{R}}$ by the second loop is $2^{\ell_{\text{ext}}}$, which is also the ulp of the correction constant. If we attempt to reduce the constant by $2^{\ell_{\text{ext}}}$, then we must also reduce t by one, i.e., add back one computed bit to the bit heap. As this bit is non-constant, this is likely to be more expensive. In this sense, this heuristic is optimal (it only remains heuristic because we do not account for the cost of the constant bits and because of the simplifying assumptions made).

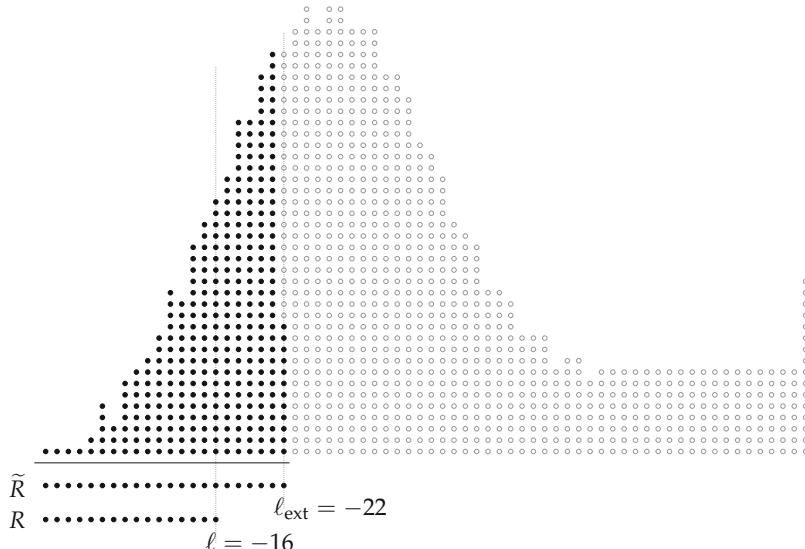


Fig. 7.18 Truncated bit heap for $X - X^3/6$ with X in $\text{ufix}(-1, -16)$, faithful to 2^{-16} (constant bits not shown).

Figure 7.18 shows the truncation that this algorithm allows on $X - X^3/6$ for an output faithful to the $\text{ufix}(-1, -16)$ format. There are several inter-

esting observations to make on this example. The first is of course that the bulk of the computation is avoided. However, it does not come cheap: there are still, in the truncated bit heap, more bits to the right of column ℓ than to the left. This was not the case in the multiplier.

We are now in a position to perform a fair comparison of this fused operator with a composition of smaller operators (each similarly truncated with love and care). We will leave this comparison as an exercise for the reader (a long and difficult one!). Our message here is that such comparison should really be performed on the basis of the truncated bit heaps, not the exact ones.

For instance, we are also only now in a position to make a more informed decision about the implementation of the division by 3. If we look back at Fig. 7.16, the second summation, the one with the infinite tail, seemed to contribute a lot to the bit heap, but we can now see that what we actually need among these bits is a flat triangle peaking at a height of 3 bits, not 8. Computing these few bits and compressing them will probably be cheaper and faster than pre-dividing them by 3.

7.2.4.5 MSB Considerations

The correction constant used so far may have the unfortunate effect of incrementing the MSB of the result compared to the non-truncated case, and the reader should be aware of this—although the authors did not encounter this issue in a practical situation so far.

For illustration, it is easy to see that this issue does not affect integer multipliers or squarers. In an unsigned multiplier, e.g., for $\text{ufix}(-1, \ell)$ inputs, the largest input is $1 - 2^\ell$. The largest possible product is $1 - 2 \cdot 2^\ell + 2^{2\ell}$. A faithful truncated multiplier or squarer entails an error bounded by 2^ℓ ; therefore it does not impact the MSB of the result. For signed inputs, the problem is even simpler since the MSB of the result is used only for the single product -1×-1 (see Sect. 8.1).

Similar proof may be conducted for more complex truncated bit heaps, sometimes taking the value of the correction constant into account.

7.2.4.6 Refinements of the Method

We have considered that the worst-case truncation error happens when all the truncated bits are equal to 1. However, for some bit heaps, it may happen that no input combination ever sets to one all the truncated bits together. Consider, for instance, the bit heap computing $Z - Z^3/6$: we have minus signs in (7.8), which are implemented (see Sect. 7.2.2) as complemented bits plus constant bits. Therefore, in (7.8), when an input bit x_i is 1, it means in the

bit heap a 1 for $\sum_{\ell \leq i \leq m} 2^i x_i$, but a 0 for $-\sum_{\substack{\ell \leq i < j \leq m \\ j \leq -1}} 2^{3i-1-2j} x_i$; it is impossible that all the bits in the bit heaps of Fig. 7.15 are equal to 1 together. In this example, it will turn out that all the *truncated* bits are negative, so the worst case does happen for $X = 0$. Still, it is easy to imagine a situation where we have occurrences of both x_i and \bar{x}_i in the truncated region. This was just an illustration; more subtle correlations may happen, including ones that come from the context (e. g., when certain combinations of the inputs cannot happen).⁴

The previous approach may be suboptimal in the presence of such correlations (however, it is always safe). In a programmatic approach, if these correlations are well defined, they can probably be taken into account, but beware that such correlations may also remove some of the hypotheses used in the implementation of the truncation algorithm. In particular, in a weighted sum of ones, we have a lexicographic order in (ℓ_{ext}, t) that enables the two-loop approach. This order may no longer exist in a weighted sum of correlated bits. The optimization program must then take a more exhaustive approach.

Another possible refinement is the use of non-constant correction terms instead of our constant C . It has been much studied in truncated multiplier literature [KS97; MTV06; Pet+10; KH11; DRC14]. This may improve the statistical behavior of the error [Pet+10]. However, for the “computing just right” paradigm of this book, based on a worst-case analysis, the benefit of variable correction terms is unclear. Besides, constant terms have the advantage that they can be pre-added in bit heaps involving several multipliers, which dilutes their cost.

The remainder of this chapter addresses the compression of a bit heap.

7.3 Compressor Tree Synthesis

This section addresses the generation of compressor trees. A compressor tree inputs a bit heap and returns a bit heap of height at most 2. This output bit heap must then be input to a final adder, which can be a fast one or a low-resource one, depending on the demands of the application: we refer the reader to Chap. 5 for details on how to implement this final adder.

First, Sect. 7.3.1 presents simple but effective historical algorithms to design compressor trees out of full adders. But it is also possible to use elementary compressors that are coarser and more efficient than full adders: Sect. 7.3.2 is a review of such elementary compressors, both for application-specific integrated circuits (ASICs) and field programmable gate arrays (FPGAs).

⁴ If it is possible to manage such correlations by a rewriting that reduces the bit heap size, it is of course better.

GAs). A general algorithm exploiting these higher-order compressors is then presented in Sect. 7.3.3.

7.3.1 Basic Compression Algorithms

The historical algorithms presented here rely only on FA and HA cells. As introduced in Sect. 5.2.1, an FA computes the sum of three bits of equal weight and producing a 2-bit vector, while the HA only has two inputs. Hence, the HA is not really a compressor as it does not remove bits, but it is useful to move bits from LSB to MSB in compressor trees.

7.3.1.1 Dadda's Algorithm

Dadda observed that the compression of a bit heap of a certain height, using full adders, requires a minimum number of levels. His heuristic [Dad65] is to determine this minimum number; and then use at each level the minimal number of full adders and half adders to just keep the minimum levels possible. This algorithm is believed to be optimal with respect to the number of full adders while having the minimum number of levels [Par10]. However, a formal proof for this is missing to the best of the authors' knowledge.

To obtain the minimal number of levels, one has to understand that one row of full adders takes three rows of bits and produces two rows of bits. Thus, it can compress a bit heap by at most a factor $\frac{3}{2}$. Starting from the output of the compressor tree, where (by definition) the maximum column height is two, one level of full adders allows a column height of $\lfloor 2 \cdot \frac{3}{2} \rfloor = 3$. With two levels of full adders, the maximum column height becomes $\lfloor 3 \cdot \frac{3}{2} \rfloor = 4$, etc. In general, let us note m_l the maximum height that can be compressed by l levels of full adders, where the levels are numbered in such a way that $l = 0$ denotes the output level, i. e., $m_0 = 2$. The sequence (m_l) can be recursively computed by

$$m_l = \lfloor 1.5m_{l-1} \rfloor \text{ with } m_0 = 2. \quad (7.25)$$

This recursion leads to a sequence, sometimes called *Dadda sequence*, which is shown in Table 7.1.

Table 7.1 First elements of the Dadda sequence.

l	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
m_l	2	3	4	6	9	13	19	28	42	63	94	141	211	316	474

Dadda's algorithm is now defined as follows:

1. Find the smallest L such that m_L is greater or equal to the maximum column height of the given bit heap. Initialize the current level to $\ell = L$ (the input level).
2. For each column i , starting with the LSB: if the height of the column i is larger than $m_{\ell-1}$, apply the minimal number of compressors (full adders and at most one half adder) that reduce the column height to $m_{\ell-1}$. The carry bits produced by these compressors must be added to column $i + 1$, to be counted in its height while processing the current level ℓ . However, they should not be compressed before level $\ell - 1$, so as to avoid a carry propagation within the level.
3. Set $\ell \leftarrow \ell - 1$ and repeat step 2 until no column with more than two bits exists (then, $\ell = 0$).

Dadda's algorithm can be seen as an algorithm in which the compressors are scheduled in an as late as possible (ALAP) strategy.

Consider the example shown in Fig. 7.19, which shows the bit heap of an unsigned 8×8 bit multiplier using (7.3). The column height at the input level is eight; hence, according to Table 7.1, there are at least $L = 4$ stages necessary to build a compressor tree as $m_4 = 9 \geq 8 > m_3 = 6$. We start with applying compressors to level $\ell = 4$ producing bits for level $\ell = 3$. We scan the bit heap of $\ell = 4$ from LSB to MSB and search for columns higher than $m_{L-1} = m_{4-0-1} = m_3 = 6$ bits and apply full adders and half adders as necessary to reduce the column height to just 6 bits. On level $\ell = 4$, the rightmost half adder is necessary in column 6 to reduce the column height from 7 to 6. A half adder and a full adder are necessary in column 7 to reduce the column height from 9 (8 input bits plus the carry of the half adder of column 6) to a column height of 6, etc. As a result, 0 full adders and 0 half adders are necessary for this example.

7.3.1.2 Reduced Area Algorithm

While Dadda's algorithm provides the least amount of half adders and full adders per stage, the number of bits in each stage is maximal due to the ALAP philosophy. This leads to a high amount of interconnect wires between the stages or, in case the stage is pipelined, a high amount of flip-flops (FFs). Besides, the final adder is larger than necessary in Dadda's algorithm: looking back at Fig. 7.19, it would have been possible to use one more half adder at column 1 of level 3 to start the carry propagation, removing column 1 from the final adder. More columns could be removed this way.

Here, the reduced area (RA) algorithm of Bickerstaff et al. [BSS93] provides an alternative. It follows an as soon as possible (ASAP) strategy, reducing the number of bits in each stage but with a slightly higher cost in full

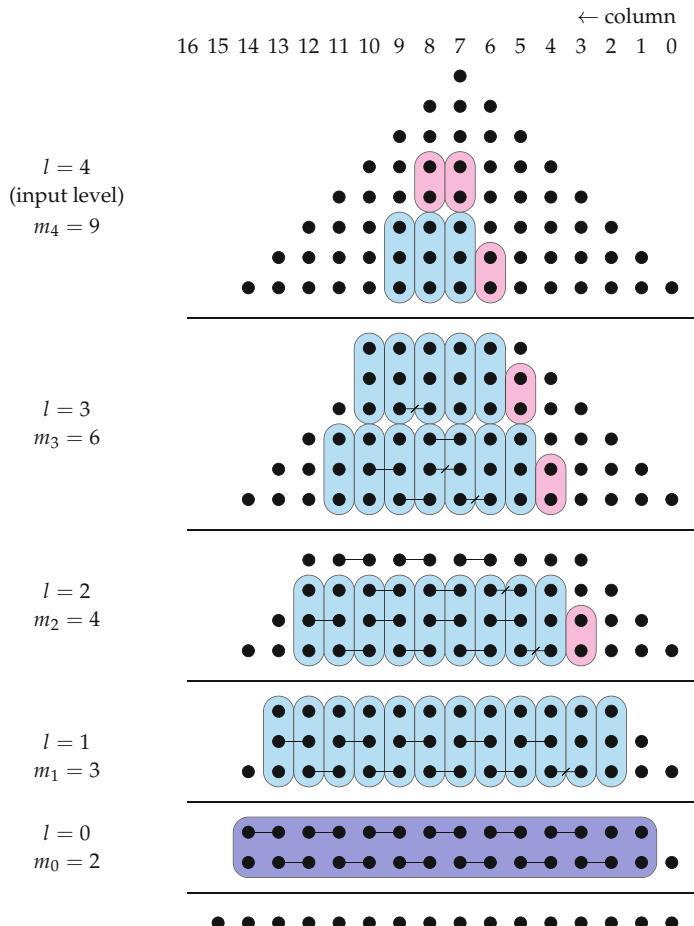


Fig. 7.19 Compressor tree design using Dadda's algorithm.

adders. This algorithm also includes a special rule to reduce the word size of the final adder. The RA algorithm is defined as follows:

For each level $l = L \dots 0$:

1. For each column, the maximum number of full adders is used such that all inputs are connected.
2. Half adders are used only (a) when required to reduce the number of bits in a column as required by the Dadda series or (b) to reduce the rightmost column containing exactly two bits.

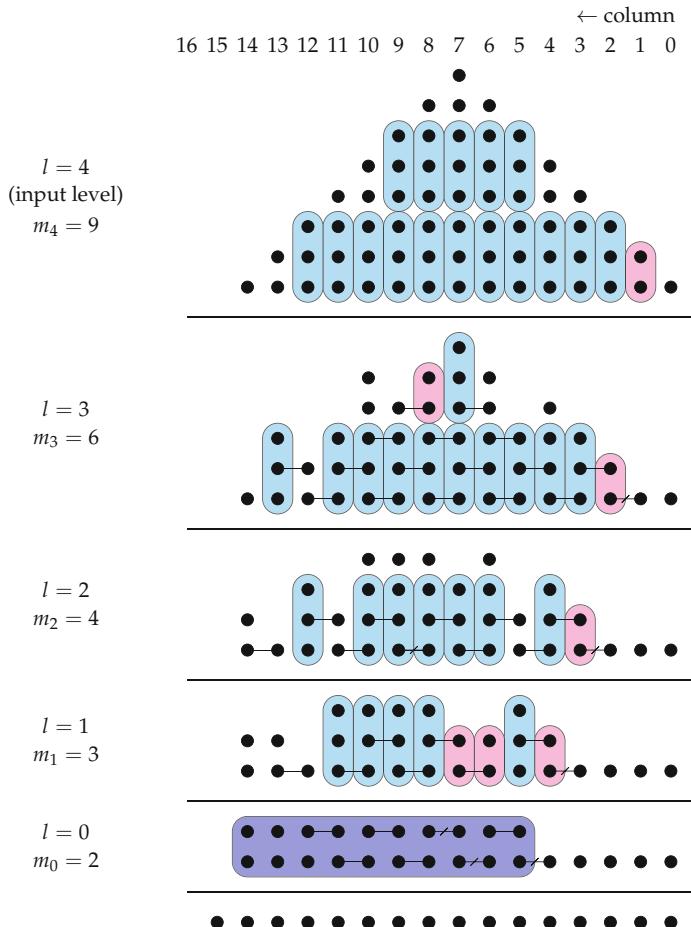


Fig. 7.20 Compressor tree design using the RA algorithm.

Figure 7.20 shows the compressor tree design of the same bit heap problem already used in Fig. 7.19 but now with the RA algorithm. As can be seen, full adders are used whenever possible. Half adders are only used to assure that the column height is not exceeded (applied in column 8 for $l = 3$ and columns 6 and 7 for $l = 1$) or to reduce the length of the final adder (applied in columns 1–4 for $l = 4 \dots 1$, respectively).

A comparison of the resources used by both algorithms is given in Table 7.2. It shows the resulting FA and HA counts, the length of the final adder as well as the FF count, assuming that a pipeline stage is inserted after every full adder (which is most probably too often but shows the extreme case). It can be observed that while the Dadda tree requires fewer full

adders (FAs) and half adders (HAs), the RA tree requires the smaller final adder and a significant lower amount of FFs when pipelined.

Table 7.2 Resource comparison of the example compressor trees obtained by Dadda (Fig. 7.19) and RA (Fig. 7.20).

Method	FA	HA	Final adder length	FF (fully pipelined)
Dadda	0	0	14	-80
RA	0	0	10	-80

7.3.2 A Bestiary of Compressors

Half adder and full adder constitute a minimal set of compressors, but it can be more efficient to extend this set with larger compressors which generalize them. This will depend on the target technology, e. g., ASIC or FPGAs.

7.3.2.1 Counters

The full adder is an instance of a family of compressors called *counters* or sometimes *parallel counters* [Swa73; Vee+07; Swa04]. A $(p; q)$ counter is a circuit that has p input bits, counts the number of bits which are one among these inputs, and outputs the count on q bits. From this definition, the number of output bits is $q = \lceil \log_2(p + 1) \rceil$. In the dot diagram, a $(p; q)$ counter removes a column of p bits and replaces it with a row of q bits. Thus, the full adder is a $(3; 2)$ counter, and the HA is a $(2; 2)$ counter.

For a given q , it is possible to compress up to $2^q - 1$ bits, and this is the most efficient in terms of compression. The next $(2^q - 1; q)$ counters are the $(7; 3)$ and $(15; 4)$ counters.

Any counter can be implemented by combining full adders. For instance, the $(7; 3)$ counter can be implemented using 4 full adders in three levels (this is left as an exercise for the reader; the solution can be found in the literature [MPS91; FF17]). However, on ASIC, it is also possible to design more efficient counter architectures by optimizing them directly at the gate level [JS94; FF17] or even at the transistor level [Swa04; Vee+07]. It is a win if area and/or delay are smaller than a realization using full adder and half adder cells.

Of course, a $(p; q)$ counter only makes sense when the height of the bit heap is larger than p : they cannot be used in the last compression stages. Large counters are therefore all the more useful as the bit heap to compress is large.

7.3.2.2 Row Compressors

A very good compressor trade-off on ASIC is the 4:2 row compressor [Wei81; PP01] depicted in Fig. 7.21. The elementary 4:2 compressor cell actually has five inputs and three outputs, but is intended to be used in a row with a connection of the neighboring cell through the k_i signal. This does not entail a carry propagation, though, as Fig. 7.21a clearly shows.

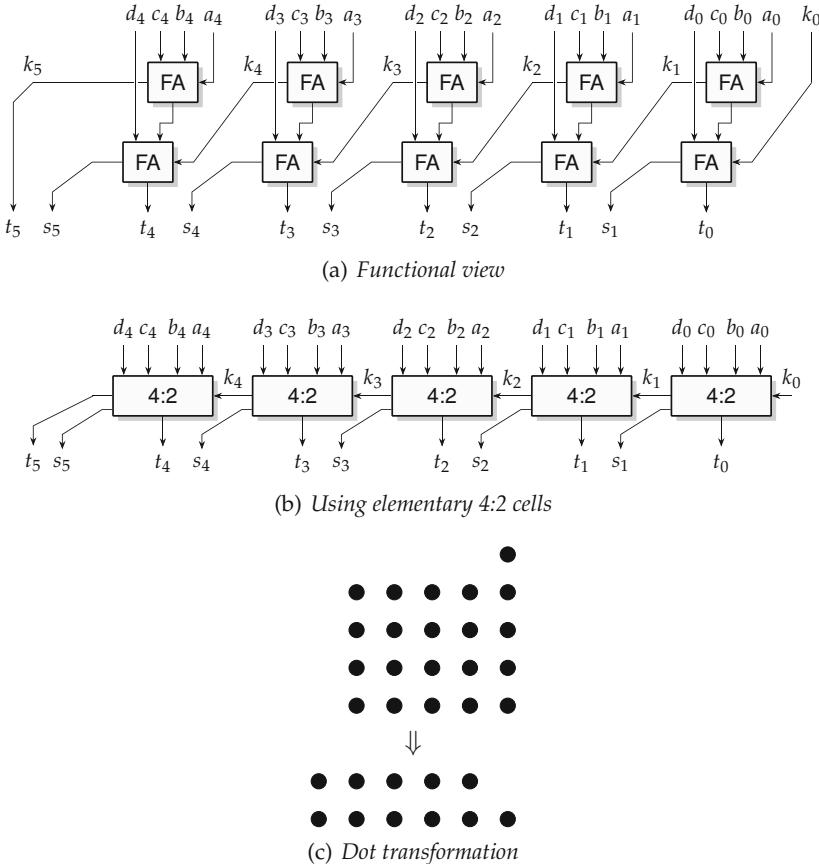


Fig. 7.21 A 4:2 compressor row. All the indices correspond to bit positions.

A 4:2 row compressor of size n compresses $4n + 1$ bits into $2n + 1$, as illustrated by Fig. 7.21c for $n = 4$.

Again, using a 4:2 row compressor only has some benefit over using full adder cells if the elementary 4:2 compressor cell visible on Fig. 7.21b is optimized somehow [LN96; PP01; CGZ04] at the gate or transistor level. An

advantage of the 4:2 compressor row over large counters is then that it can be used up to the last compression stage.

The 4:2 row compressor has been generalized to 5:2 [PP01; CGZ04] and 7:3 [MPS91] row compressors. The elementary 7:3 compressor cell has nine inputs and five outputs, with two carries being transmitted to the neighboring cell. It is possible to generalize it further, but there is a diminishing return here.

7.3.2.3 Generalized Parallel Counters

Generalized parallel counters [Meo75; SKG77] (also called multicolumn counters [EL04]) are generalizations of the counters introduced above. In contrast to them, their input bits may have different weights and thus be located in different columns.

A generalized parallel counter (GPC) is commonly denoted as a tuple $(p_{n-1}, p_{n-2}, \dots, p_0; q)$, where p_j represents the number of input bits of weight 2^j and

$$q = \left\lceil \log_2 \left(\sum_{j=0}^{n-1} 2^j p_j + 1 \right) \right\rceil \quad (7.26)$$

is the number of output bits. A $(3, 5; 4)$ GPC, for example, computes the sum of three input bits of weight two plus five input bits of weight one. The output is a number in the range $0 \dots 2 \cdot 3 + 5 = 11$ and is represented by a 4-bit vector.

7.3.2.4 Generalized Parallel Counters for FPGAs

Carry-save arithmetic has long been regarded as unsuitable on FPGAs. This was due to the fast dedicated carry chains found in modern FPGAs, while carry-save logic had to use the significantly slower generic routing fabric. However, it was first shown by Parandeh-Afshar et al. [PBI08a; PBI08b] that significant delay improvement could be obtained by using compressor trees on FPGAs. The key was to study GPCs which can be efficiently implemented in FPGAs, using their large look-up tables (LUTs) and fast carry logic.

GPCs can utilize the LUTs of the FPGA much better than the full adder does. A key insight is that an FPGA with 6-input LUT can implement very efficiently any GPC that has fewer than six inputs, just by tabulating all of its results in the LUT. Useful examples are the $(1, 5; 3)$ GPC or the $(6; 3)$ counter, which will consume 3 LUT6 each. Having the choice between these two compressors is an optimization opportunity, as it allows for different patterns in the dot diagram that may fit better to the given bit heap. How-

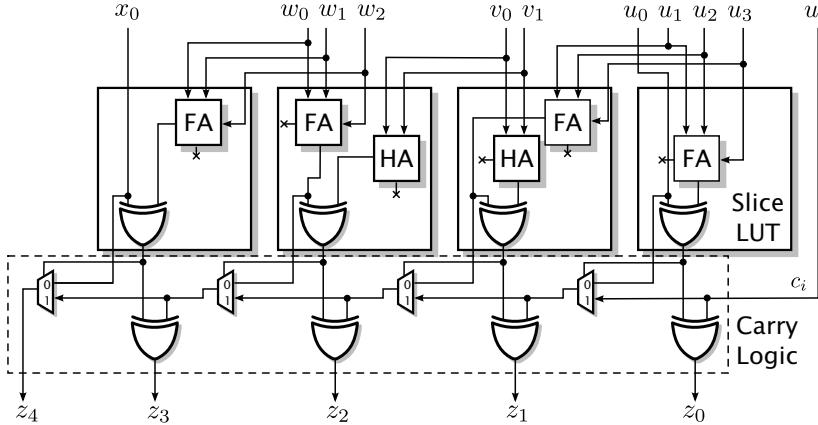


Fig. 7.22 Implementation of a (1,3,2,5;5) GPC targeting AMD FPGAs.

ever, the compression algorithm has to be able to exploit this opportunity: it will be the case of the algorithm of Sect. 7.3.3.

But FPGAs also provide fast carry chains, and these can be utilized to increase the efficiency of the GPC. Figure 7.22 shows an example of a (1,3,2,5;5) GPC that utilizes the four LUTs as well as the fast carry chain of a single AMD Slice.

An important metric to evaluate GPCs is their *efficiency* [Bru+13] (also called *area degree* in [Par+11]) which is defined as the quotient of the number of removed bits and the number of required LUTs k

$$E = \frac{b_i - b_o}{k} = \frac{\delta}{k} \quad (7.27)$$

where b_i and b_o denote the number of input and output bits, respectively. High-efficiency GPCs typically have the property that their input count may be highly irregular, like the (1,3,2,5;5) GPC of Fig. 7.22.

Another GPC metric is the *compression ratio* [PBI08a] (also called *compression factor* [Bru+13]), which is defined as the ratio of input to output bits:

$$\gamma = \frac{b_i}{b_o}. \quad (7.28)$$

Early work indicated that a higher compression ratio leads to fewer stages in the compressor tree [PBI08a; Bru+13]. However, recent work [KK18] showed that the efficiency metric is more effective without sacrificing the stage count in most of the cases.

Table 7.3 shows a comparison of different GPCs and counters used in previous work that are based on pure LUT implementations (top part) as well as GPCs and counters that utilize the fast carry chains (middle part).

Table 7.3 High-efficiency GPCs, counters, and adders divided into LUT-based GPCs/-counters (top), LUT+carry chain-based GPCs/counters targeting AMD FPGAs (middle), and row adders targeting AMD FPGAs (bottom).

GPC/ row adder	Ref.	#LUT6 (k)	Efficiency ($E = \delta/k$)	Delay
(3;2)	[Bru+13]	1	1	$\tau_L \approx \tau$
(6;3)	[Bru+13]	3	1	$\tau_L \approx \tau$
(1,5;3)	[Bru+13]	3	1	$\tau_L \approx \tau$
(7;3)	[Yua+19]	2	1.33	$\tau_L + 2\tau_{CC} \approx \tau$
(2,3;3)	[KZ14a]	2	1	$\tau_L + 2\tau_{CC} \approx \tau$
(1,4,1,5;5)	[KZ14a]	4	1.5	$\tau_L + 4\tau_{CC} \approx \tau$
(1,4,0,7;5)	[KZ14a]	4	1.75	$\tau_L + 4\tau_{CC} \approx \tau$
(1,3,2,5;5)	[KZ14b]	4	1.5	$\tau_L + 4\tau_{CC} \approx \tau$
(6,2,3;5)	[Pre17]	4	1.5	$\tau_L + 4\tau_{CC} \approx \tau$
(6,0,7;5)	[Yua+19]	4	2	$\tau_L + 4\tau_{CC} \approx \tau$
(6,1,5;5)	[Pre17]	4	1.75	$\tau_L + 4\tau_{CC} \approx \tau$
(2,1,1,7;5)	[Yua+19]	4	1.5	$\tau_L + 4\tau_{CC} \approx \tau$
2-input add.		k	1	$\tau_L + k\tau_{CC}$
Ternary add.		k	$2 - \frac{2}{k}$	$2\tau_L + \tau_R + k\tau_{CC} \approx 3\tau + k\tau_{CC}$
4:2 compressor	[KZ14a]	k	$2 - \frac{1}{k}$	$\tau_L + k\tau_{CC}$

As expected, the GPCs optimized for the carry chains of the middle part provide the best efficiency. The last column shows the corresponding delay of the compressor. It consists of LUT delays τ_L , routing delays τ_R , and carry chain delays τ_{CC} . As the LUT delay typically is similar to the delay of a local routing, we can assume $\tau_L \approx \tau_R \approx \tau$. As the carry-propagation is typically orders of magnitudes lower than τ , we can ignore this delay for small carry chains. The dot transformations (i.e., the dots they remove/produce) are shown for all these compressors in Fig. 7.23.

7.3.2.5 Row Compressors for FPGAs

Thanks to fast carry logic, row compressors on FPGA can use a carry propagation. Adders and ternary adders can therefore be considered row compressors on FPGAs (ternary adders have been already discussed in detail in Sect. 5.4.4).

Applying the efficiency metric defined in (7.27) to row compressors, we get an efficiency that depends on the row width k as shown in the lower part of Table 7.3. The common 2-input adder achieves a constant efficiency of just 1.

In contrast to that, the ternary adder provides a high efficiency, reaching $E = 2$ for output word size $k \rightarrow \infty$. However, as already discussed in Sect. 5.4.4.1, the ternary adder is typically slower than the 2-input adder.

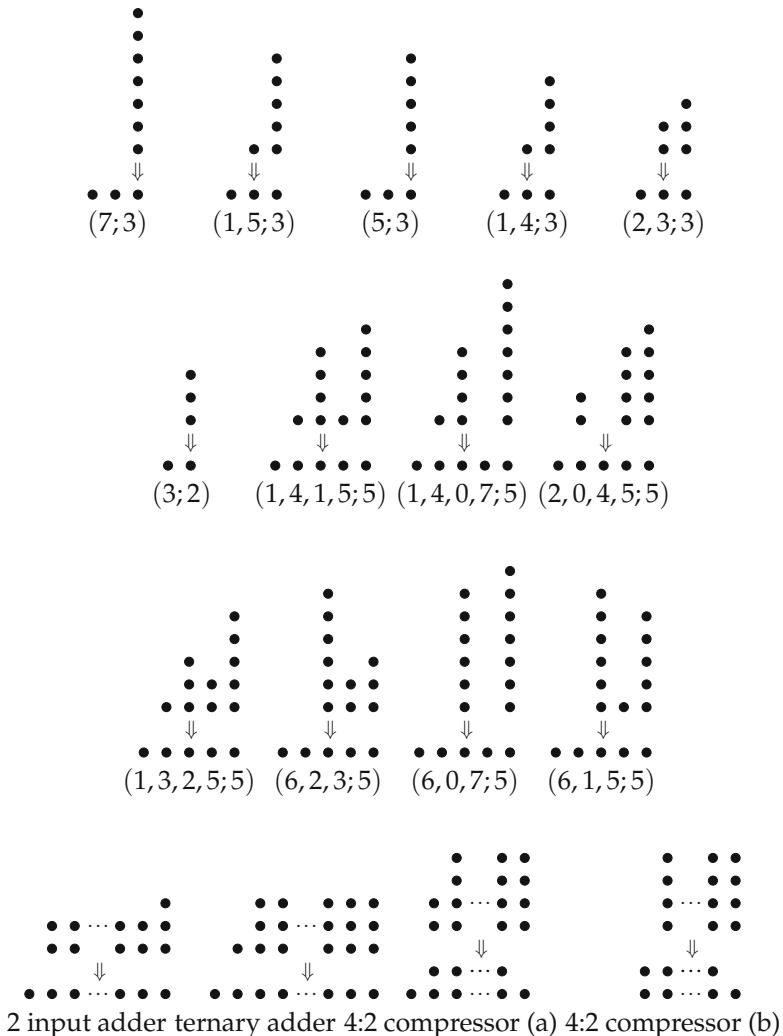


Fig. 7.23 Dot transformations for the GPCs and row adders from Table 7.3.

Thus, it is the slowest compressor among the considered ones due to the slow routing of carries.

To get rid of the delay, the carry does not have to be routed within the same adder. It may be saved, following the concept of carry-save arithmetic. This leads to one additional output row but also one additional input row (which is now free as no carry has to be processed). We get a 4:2 compressor, whose efficient mapping to the slice structure of AMD FPGAs proposed in [KZ14a] is shown in Fig. 7.24. Note that in contrast to the conventional 4:2

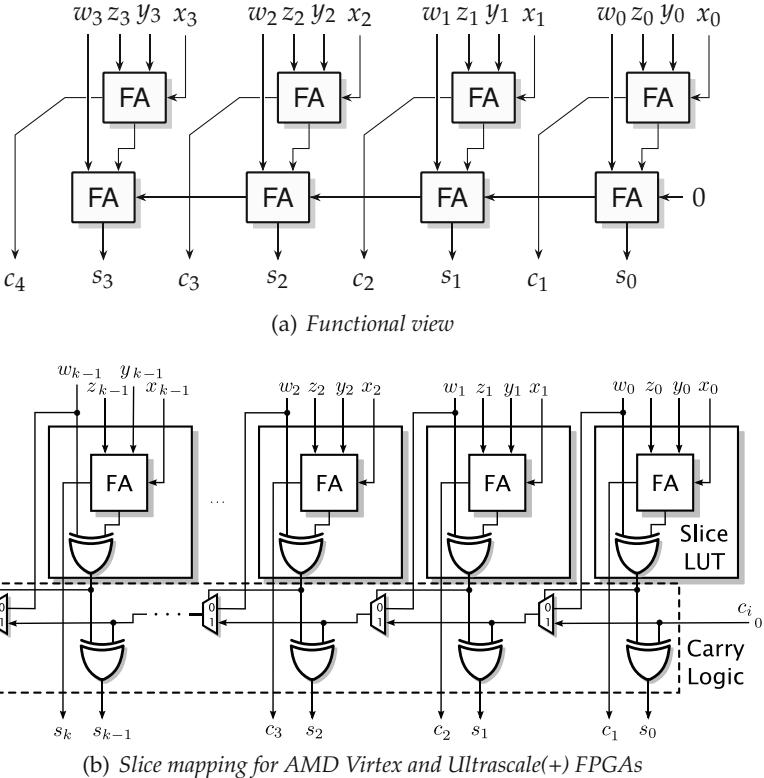


Fig. 7.24 Efficient 4:2 row compressor exploiting fast carry logic on FPGAs.

compressor where any carry propagation is avoided (see Fig. 7.21a), the 4:2 compressor in Fig. 7.24a explicitly utilizes the fast carry chain of the FPGA. It reaches a slightly better efficiency than the ternary adder, with a much shorter delay.

7.3.3 Compressor Tree Synthesis Using Arbitrary Compressors

This section discusses different ways to design compressor trees that, contrary to the basic compression algorithms presented in Sect. 7.3.1, can exploit compressors of arbitrary shapes such as those of the previous section. We start with a very generic and fast heuristic and then present an integer linear programming (ILP) formulation to optimally solve the bit heap compression problem with respect to a minimum number of stages and minimum amount of logic resources. As the ILP-based approach does not scale

to large problem sizes, we also present a combined approach that uses the heuristic for partial optimization for problem reduction while leaving the critical parts to the ILP solver.

All the methods discussed here have been originally published in [KK18] and have the same interface: they take as argument a bit heap to compress, a list of compressors to use (depending on the target device), and the number of inputs of the final adder:

$$I = \begin{cases} 2 & \text{for 2-input final add;} \\ 3 & \text{for ternary final add.} \end{cases} \quad (7.29)$$

A last parameter that is only used for the combined approach is the minimum effective efficiency for each stage s ($\overrightarrow{\text{eff}}_{\min}(s)$).

7.3.3.1 A Universal Heuristic

The heuristic explained next is based on the heuristic proposed by Parandeh-Afshar et al. [PBI08a; Par+11], extended by an improved compressor selection and a better scaling for row compressors and large GPCs. The pseudo code of the heuristic is given in Listing 7.2.

In line 3, the compressors are ordered by their efficiency as defined by (7.27). For the row adders, the maximum efficiency is assumed for the ordering ($k \rightarrow \infty$ in Table 7.3) at this step.

GPCs which can be implemented by another GPC at the same or lower cost are removed at this step. For example, for an AMD target, the (7;3) GPC of Table 7.3 can realize the (6;3) GPC with a better cost. Hence, the (6;3) GPC is excluded from the list for this target.

The algorithm works in compression stages, starting from the input stage ($s = 0$). The outer loop of the algorithm iterates over stages. A stage is processed once all its dots have been removed (Line 30).

For each stage, the columns are first ordered by their height (Line 9). In case of equal height the column with the lower weight is preferred.

Now, the compressors are examined in the next inner loop from highest to lowest efficiency. In the most inner loop, the *effective efficiency* is computed for each compressor and column (Line 15). To compute the effective efficiency, only the actually connected inputs b_i are counted in (7.27) when applied to a specific column. It is a key idea of the heuristic to select the compressors by their effective efficiency.

For the row compressors, the effective efficiency is a variable of the row width k . Here, the row width is increased from $k = 2$ to the maximum number of columns, and the highest efficiency obtained is returned.

The compressor with best efficiency is saved in lines 16–20.

If the selected compressor is a GPC and the achieved *effective* efficiency is equal to its efficiency, i. e., all its inputs are used, the loop can be terminated

```

1 heuristicImp(bitHeap, compressorList, I,  $\overrightarrow{\text{eff}}_{\min}$ ) :
2
3 compressorList = orderByEfficiency(compressorList)
4 s = 0
5 cols[0] = bitHeap
6
7 while max(cols[s]) > I do
8   do
9     cols[s] = orderColumnsByHeight(cols[s])
10    effbest = 0
11
12    foreach comp in compressorList do
13      if(getEfficiency(comp) ≤ effbest) break
14      foreach c in cols[s] do
15        eff = evalEffectiveEfficiency(comp, c)
16        if eff > effbest do
17          effbest = eff
18          compbest = comp
19          cbest = c
20        end
21        if(type(comp) ≡ GPC and
22           getEfficiency(comp) ≡ effbest) break
23      end
24    end
25
26    if(eff <  $\overrightarrow{\text{eff}}_{\min}(s)$ ) break
27
28    cols[s] = removeDots(cols[s], cbest, compbest)
29    cols[s + 1] += genDots(cols[s], cbest, compbest)
30  until all dots of cols[s] have been removed
31
32  s ← s + 1
33 end
34
35 generateFinalCPA(cols[s - 1])

```

Listing 7.2 Bit heap compression heuristic.

(Line 22): no better GPC will occur in the sorted list. For the same reason, we can terminate the next outer loop when the efficiency of the next compressor is lower or equal than the best efficiency found so far (Line 13).

Once a compressor is found, the corresponding dots are removed, and dots of the succeeding stage are generated (Lines 28–29). Line 26 provides a possibility to skip compressors which have an efficiency less than the user-defined minimum effective efficiency for stage s ($\overrightarrow{\text{eff}}_{\min}(s)$). This feature is only used in the combined approach which is introduced below in Sect. 7.3.3.3. For the heuristic, it is set to $\overrightarrow{\text{eff}}_{\min} = (0, \dots, 0)$.

To ensure that all the dots of one stage are eventually transferred, so that the outer loop progresses, a *pseudo* (1; 1) GPC is always included in the set of compressors. In a combinatorial compressor tree, it represents a simple wire

with zero cost, but the cost can be different in case of a pipelined compressor tree as it will be discussed below.

As the heuristic follows a greedy approach, it is fast and requires only a few seconds for even large bit heaps. However, although the heuristic typically finds good solutions, it usually fails to find the optimal solution. An ILP-based approach to the same problem is presented next to find the optimal solution, but it is limited to relatively small problem sizes.

7.3.3.2 ILP-Based Optimal Bit Heap Compression

Several works addressed the optimization of compressor trees using integer linear programming (ILP) [PBI08b; MKM11; MKM13; KK18]. They differ in the way the ILP variables describe the problem and in their optimization goals (LUT cost in combinatorial compressor trees, number of FFs in pipelined compressor trees, or a combination of both).

The ILP formulation presented here can be used for both optimization goals. All used variables and constants are summarized in Table 7.4. The main idea is to count the number of bits (dots) in each column $c = 0 \dots C$ for each stage $s = 0 \dots S$ using variables $N_{s,c}$. The initial problem is thus defined by setting for each column c the number $N_{0,c}$ of bits in stage 0 (the input stage) to the height of the column in the input bit heap to be compressed.⁵ Next, the integer variables $k_{s,e,c}$ denote how many GPCs of type $e = 0 \dots E - 1$ are applied in stage $s = 0 \dots S - 1$ and column $c = 0 \dots C - 1$. Each compressor e is characterized by the number of input bits $M_{e,c}$ that are removed and the number of output bits $K_{e,c}$ that are generated in column c , respectively. For example, if compressor $e = 4$ is a $(1, 5; 3)$ GPC, then $M_{4,0} = 5$, $M_{4,1} = 1$, and $K_{4,0\dots 2} = 1$. With this notation, compressors with more than one output bit per column can also be represented.

The number of stages S is considered here as a constant, because it turned out to be faster to run several optimization runs with a fixed S than to treat S as a variable of the ILP model. The reason is that most solvers determine relatively quickly that the problem is infeasible when S is too low for a given problem. With S a constant in the ILP problem, we simply run the ILP solver several times, for increasing values of S starting with $S = 1$, until a feasible solution is found. The number of stages needed to compress a bit heap remains typically small (it grows as the logarithm of the height of the initial bit heap, since it is bounded in any case by the Dadda sequence); therefore this outer loop remains fast.

Using the variables and constants described in Table 7.4, the following ILP formulation describes a compressor tree of minimal cost:

⁵ If input bits arrive late or in a different cycle [Bru+13], this can be easily extended to setting $N_{s,c}$ for other stages as well.

Table 7.4 Variables (top) and constants (bottom) used in the compressor tree ILP formulation.

Variable/ Constant	Meaning
$N_{s,c} \in \mathbb{N}$	Number of bits in stage s and column c
$k_{s,e,c} \in \mathbb{N}$	Number of compressors of type e in stage s and column c
$c_e \in \mathbb{R}$	Cost (e.g., in LUTs) of compressor e
$M_{e,c} \in \mathbb{N}$	Number of bits removed from compressor e in column c
$K_{e,c} \in \mathbb{N}$	Number of bits generated from compressor e in column c
$E \in \mathbb{N}$	Number of compressing elements
$C \in \mathbb{N}$	Maximum number of columns
$C_e \in \mathbb{N}$	Maximum number of columns of compressor e
$S \in \mathbb{N}$	Number of stages

$$\text{minimize } \sum_{s=0}^{S-1} \sum_{c=0}^{C-1} \sum_{e=0}^{E-1} c_e k_{s,e,c}$$

subject to

$$\mathcal{C}1 : N_{s-1,c} \leq \sum_{e=0}^{E-1} \sum_{c'=0}^{C_e-1} M_{e,c+c'} k_{s-1,e,c+c'} \quad \text{for } s = 1 \dots S, c = 0 \dots C-1$$

$$\mathcal{C}2 : N_{s,c} = \sum_{e=0}^{E-1} \sum_{c'=0}^{C_e-1} K_{e,c+c'} k_{s-1,e,c+c'} \quad \text{for } s = 1 \dots S, c = 0 \dots C-1$$

$$\mathcal{C}3 : N_{S,c} \leq I$$

The objective is to minimize the resource cost. For that, the number of used GPCs per stage and column $k_{s,e,c}$ are weighted by their corresponding cost c_e .

Now, the first constraints ($\mathcal{C}1$) ensure that all bits in each column and stage except the output stage are connected to inputs of compressors. The fact that there may be more compressor inputs than bits available to compress is considered by the \leq -relation. Constraints $\mathcal{C}2$ simply compute the number of bits produced by compressors which are taken as input to the next stage. Finally, the column height of the output stage ($s = S$) is constrained by $\mathcal{C}3$ to be maximally equal to the input count of the final adder I (see (7.29)).

Here again, one key element in the model is that a single-input, single-output (1;1) *pseudo* GPC is included in the set of compressors.

Support for row compressors

The ILP formulation above cannot handle row compressors, except by tabulating all different widths as GPCs, which would lead to an excessive increase in variables and constraints. This can be avoided by splitting a compressor with variable size in several dependent compressors and adding constraints on how these parts are related. For that, a row compressor is divided into three partial compressors: e_L (placed at the column with lowest significance), several e_M (placed at multiple columns in the middle), and e_H (placed at the column with highest significance). An example showing how to split the 4:2 compressor (a) of Fig. 7.23 is shown in Fig. 7.25. The lowest (rightmost) column compressor reduces four bits of the first column into one bit, corresponding to a compressor with $M_{e_{L,0}} = 4$ and $K_{e_{L,0}} = 1$. The middle compressors have four inputs and two outputs in each column ($M_{e_{M,0}} = 4$ and $K_{e_{M,0}} = 2$). The highest (leftmost) compressor has two input bits and produces two output bits in the same column and another bit in the next higher column ($M_{e_{H,0}} = 2$, $K_{e_{H,0}} = 2$ and $K_{e_{H,1}} = 1$). These partial compressors only work if they are connected in the right order (due to the internal carry propagation) which can be obtained by introducing the constraints

$$C4 : k_{s,e_H,c+1} + k_{s,e_M,c+1} = k_{s,e_M,c} + k_{s,e_L,c} \text{ for } s = 1 \dots S - 1, c = 0 \dots C - 1$$

(the $k_{s,e,c}$ is defined to be zero for columns out of range, i. e., for $c \neq 0 \dots C - 1$). These constraints ensure that each compressor of type e_L or e_M in column c has an e_M or e_H compressor to its left (column $c + 1$). This guarantees the correct shape of the row compressor.

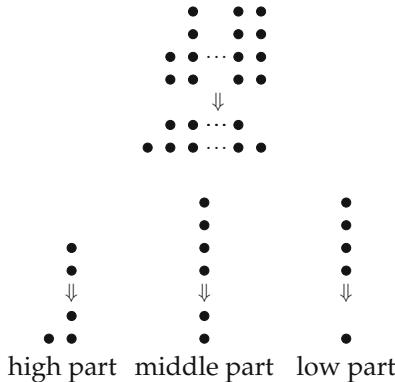


Fig. 7.25 Splitting a row compressor into three parts.

7.3.3.3 Combined Heuristic with ILP

The heuristic presented at the beginning of this section scales well to large problem sizes. The optimal ILP-based technique is limited to problems with a few hundreds of bits (see experimental results in Sect. 7.4).

The technique we present now consists of first using the greedy heuristic to reduce a large bit heap to a size that is manageable by the ILP solver, then completing the compression in an optimal way. This flow is depicted in Fig. 7.26.

Of course, even though an optimal algorithm is used, the overall algorithm remains a heuristic. However, experiments show that the selection of compressors in the early stages have only a small influence on the overall optimization quality.

For that, the heuristic part may involve the following two strategies:

1. Solving complete stages by the heuristic.
2. Solving stages partially to reduce their size. To do so, we follow the strategy that the heuristic should apply compressors only until the effective efficiency is above a certain limit ($\overrightarrow{\text{eff}}_{\min}$ in Listing 7.2).

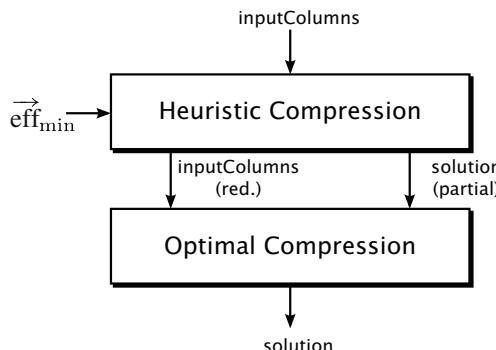


Fig. 7.26 Proposed flow of heuristic with ILP.

Both strategies can be combined in a single parameter vector $\overrightarrow{\text{eff}}_{\min}$ of size S . Each element $\overrightarrow{\text{eff}}_{\min}(s)$ can be set among one of the following three cases:

1. $\overrightarrow{\text{eff}}_{\min}(s) = 0$: The corresponding stage is completely solved by the heuristic.
2. $\overrightarrow{\text{eff}}_{\min}(s) > 0$: The corresponding stage is partially solved by the heuristic: the heuristic stops when no compressor is found that has a minimum efficiency of at least $\overrightarrow{\text{eff}}_{\min}(s)$. Hence, the corresponding stage is partially solved, and the remaining problem is solved optimally using ILP.
3. $\overrightarrow{\text{eff}}_{\min}(s) = \infty$: The corresponding stage is completely solved by the ILP method.

For example, the vector $\overrightarrow{\text{eff}}_{\min} = (0, 1.5, \infty)$ would specify that the first stage ($s = 0$) is solved by the heuristic. In stage $s = 1$ only compressors with a minimum effective efficiency of 1.5 are applied, and the remaining bits in stage $s = 1$ as well as the remaining stages ($s \geq 2$) are solved by ILP. In case that more stages are required than given in $\overrightarrow{\text{eff}}_{\min}$, missing entries are treated as being ∞ . The elements in $\overrightarrow{\text{eff}}_{\min}$ should increase monotonically as the influence of the heuristic in early stages turned out to have less effect on the objective (or can be fixed by the optimal part that follows).

As a rule of thumb, 2-3 stages can be compressed optimally in a reasonable time. Hence, once the stage count is known from a trial heuristic run, the last 2-3 stages should be set to a non-zero/infinity value.

7.3.3.4 Pipelined Compressor Trees

Although compressor trees are fast, they may not be fast enough and hence have to be pipelined. As already discussed for the RA algorithm, pipelining may be costly when bits are compressed too late. For the compressor tree algorithms using arbitrary compressors as described above, it is very easy to consider the cost of FFs used for pipelining as we introduced the (1;1) *pseudo* GPC. While the (1;1) GPC has no cost in combinatorial stages (it is just a wire), it can be assigned to the cost of a single FF in stages which include pipeline registers.

Also the cost compressors may be adjusted with additional FF cost for those stages. Here, one has to distinguish between FPGAs and ASICs. While for ASICs every compressor in a pipelined stage will have additional FF cost, this is usually not the case for FPGAs. As discussed in Chap. 4, the basic logic elements (BLEs) of FPGAs have an optional output FF that usually only has to be enabled without additional cost. This aligns well with the fact that FPGAs usually require more pipeline stages compared to ASICs due to the additional routing delays.

For the heuristic, the (1;1) GPC is always the last choice as it has a low efficiency. This means it compresses as early as possible which is good for pipelined compressor trees. The ILP formulation is of course more versatile as the problem is solved globally.

7.3.4 Some Remarks on the Final Adder

The final adder of the compressor tree may require special attention as it typically lies on the critical path, at least for ASICs. Here, one may use one of the fast adders discussed in Sect. 5.3 as final adder. However, those typically assume that all of the inputs arrive at the same time which is often not the case in compressor trees.

Many compressor trees have a Gaussian-like distribution of bits like the one in Fig. 7.15a. Here, the bits of least and highest significance arrive earliest, while the bits in the middle part arrive latest. This has been analyzed in detail for multipliers [SO96] which have a triangular shape like the one in Fig. 7.2 with a similar timing profile. Of course, this effect is less relevant in case of truncation or in pipelined compressor trees.

The timing profile can be utilized to further optimize the final adder. For example, one can split the adder into three parts. The LSBs can be added with a slow adder like the ripple-carry adder (RCA) as those bits arrive early; a fast adder is required for the middle part, while a carry-select adder can be utilized for the MSBs. Of course, this strongly depends on the distribution of bits in the bit heap as well as on the compressor tree.

7.4 Experimentations

The purpose of this section is to illustrate how optimization quality depends on the compression algorithm and how the complexity scales with the number of bits to compress. The results presented here are similar to those in [KK18], but updated with recent compressors [Yua+19] already included in Table 7.3. The experiments report the required resources as well as the stage count. The latter corresponds to the latency (in clock cycles) for pipelined compressor trees and also serves as an indicator of the total delay for combinatorial compressor trees.

Two operators are used as benchmark: a multi-input addition, implemented as FloPoCo operator `IntMultiAdder`, and an unsigned integer multiplier for FPGAs using only logic-based sub-multipliers and no DSPs (see Chap. 8), implemented as FloPoCo operator `IntMultiplier`. They represent two significantly different shapes of dot diagrams, as illustrated in Fig. 7.27.

For the `IntMultiAdder`, we set the number of inputs w identical to their word size which leads to w^2 bits where w was evaluated between 4 and 128 in steps of 4 bits. This corresponds to problem sizes of 16 to 16,384 bits. For the `IntMultiplier`, we also set both inputs to w , computing the full precision output ($2w$ bits). The input size was varied between $w = 4$ and 128, leading to problem sizes of 14 to 13,654 bits.

In the experiments, Gurobi was selected as ILP solver, with a timeout of 1 hour. The optimizations were performed for fully pipelined compressor trees, i. e., a register is placed after each GPC, leading to maximum throughput. Hence, the cost of the (1;1) *pseudo* GPC representing a FF in this case was set to 0.5 for each LUT6, as two flip-flops exist on modern FPGAs.

The optimization results for the LUT cost as well as for the latency are shown in Figs. 7.28 and 7.29. Figure 7.28 shows a comparison between the optimal and heuristic compression for problem sizes up to 1024 bit. One

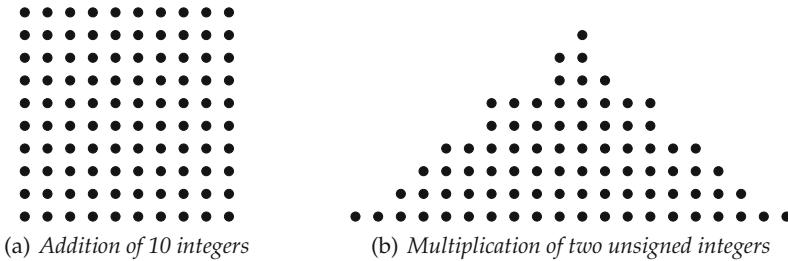


Fig. 7.27 Dot diagrams shapes used in the experiments (here for 10-bit integers).

can observe that many optimal points could not be found within the given timeout of 1 hour. In case an optimal solution was found, the difference compared to the heuristic regarding LUT cost as well as latency is clearly visible but not excessive.

The results for the full scale are shown for the heuristic in Fig. 7.29. One can observe a fairly linear trend of 0.55 LUTs per bit, which is also shown as a dashed line in Fig. 7.29a. Note that this is an improvement compared to the 0.65 LUTs per bit obtained in [KK18], thanks to the more efficient compressors of [Yua+19].

Hands on: Obtaining the bit heap benchmark circuits and setting the optimization method

The following FloPoCo calls produce and optimize the bit heaps as shown in Fig. 7.27:

```
flopoco compression=heuristicMaxEff IntMultiAdder n=10 \
    signedIn=0 wIn=10

flopoco compression=heuristicMaxEff tiling=optimal \
    IntMultiplier wX=10 wY=10 useDSP=0
```

To reproduce the designs evaluated in Fig. 7.29, the `wIn` parameter has to be varied between 10 and 32 for the `IntMultiAdder`. For the `IntMultiplier`, the input size has to be varied between 4 and 64 in steps of 4 bit.

Different compression methods can be selected by the `compression` parameter: `heuristicMaxEff` corresponds to the heuristic presented in Sect. 7.3.3.1, while `optimalMinStages` uses the optimal compression described in Sect. 7.3.3.2.

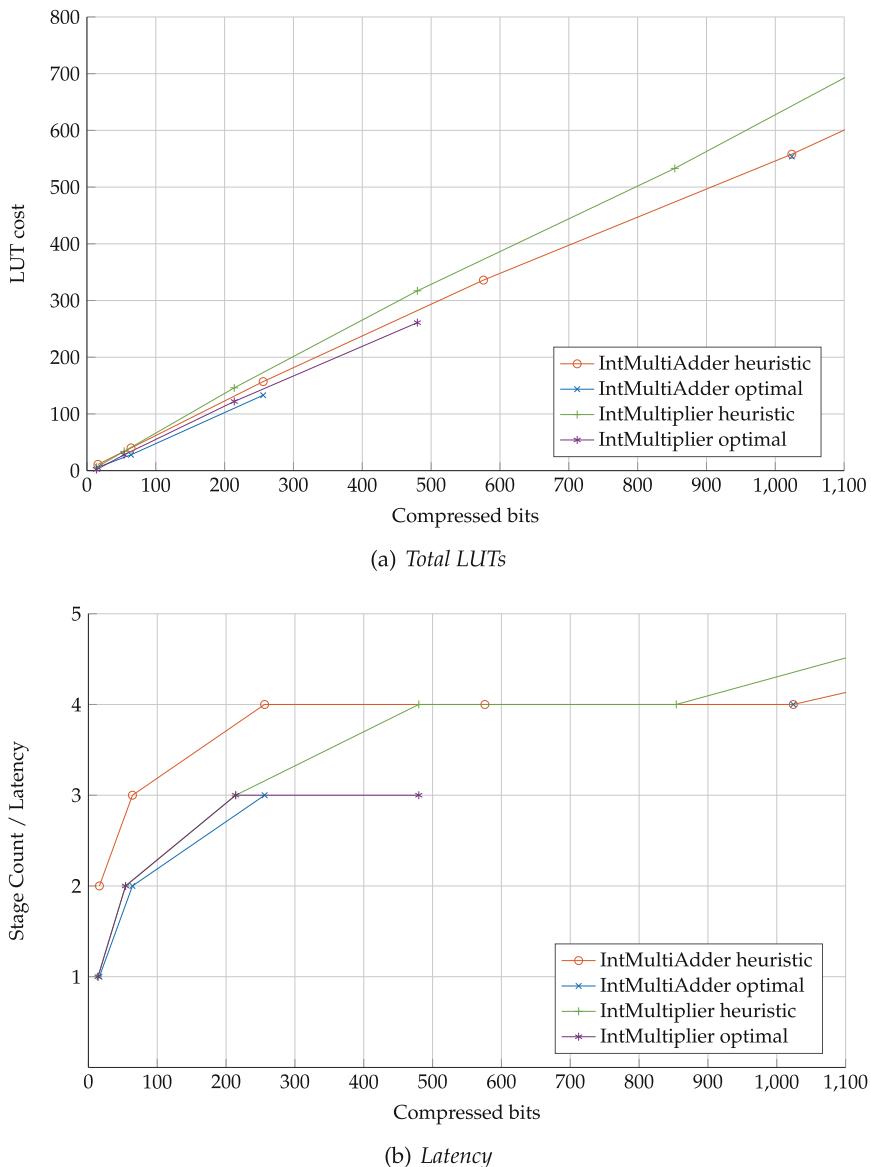


Fig. 7.28 Optimization results as LUT cost and LUT reduction for different methods for small problems up to 1000 bit.

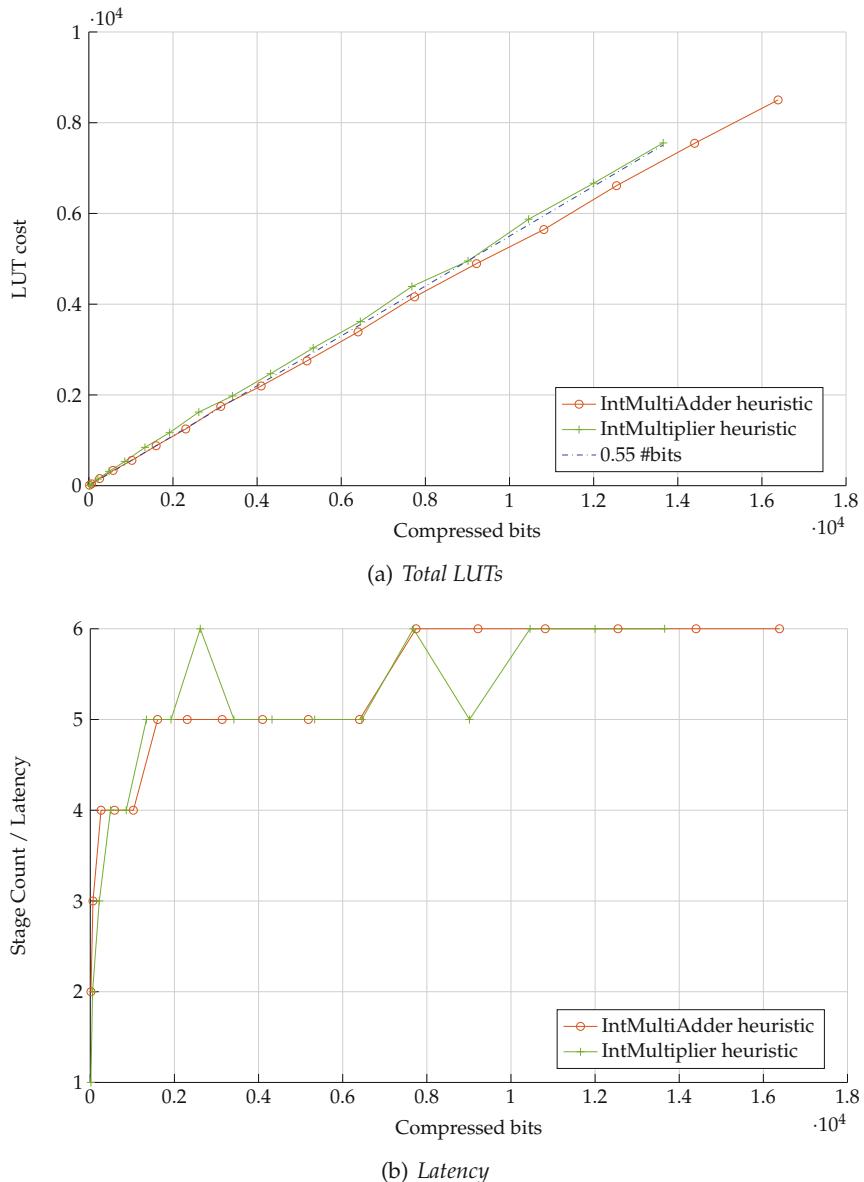


Fig. 7.29 Optimization results as LUT cost and LUT reduction for different methods for large problems.

References

- [A000292] *Tetrahedral (or triangular pyramidal) numbers*. On-Line Encyclopedia of Integer Sequences. <http://oeis.org/A000292>. 2010 (cit. on p. 165).
- [A000337] *A000337*. On-Line Encyclopedia of Integer Sequences. <http://oeis.org/A000337>. 2010 (cit. on p. 170).
- [BG09] Anton Blad and Oscar Gustafsson. “Integer Linear Programming-Based Bit-Level Optimization for High-Speed FIR Decimation Filter Architectures”. In: *Circuits, Systems, and Signal Processing* 29.1 (2009), pp. 81–101 (cit. on p. 157).
- [Bru+13] Nicolas Brunie, Florent de Dinechin, Matei Istoan, Guillaume Sergent, Kinga Illyes, and Bogdan Popa. “Arithmetic Core Generation Using Bit Heaps”. In: *Field Programmable Logic and Application (FPL)*. IEEE, 2013, pp. 1–8 (cit. on p. 761).
- [BSS93] K’Andrea C. Bickerstaff, Earl E. Swartzlander, and Michael J. Schulte. “Reduced Area Multipliers”. In: *Application-Specific Array Processors*. 1993, pp. 478–489 (cit. on p. 177).
- [BW73] Charles R. Baugh and Bruce A. Wooley. “A Two’s Complement Parallel Array Multiplication Algorithm”. In: *IEEE Transactions on Computers* C-22.12 (1973), pp. 1045–1047 (cit. on p. 231).
- [CGZ04] Chip-Hong Chang, Jiangmin Gu, and Mingyan Zhang. “Ultra Low-Voltage Low-Power CMOS 4-2 and 5-2 Compressors for Fast Arithmetic Circuits”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 51.10 (2004), pp. 1985–1997 (cit. on pp. 181, 182).
- [Che15] Chichyang Chen. “High-order Taylor series approximation for efficient computation of elementary functions”. In: *IET Computers & Digital Techniques* 9.6 (2015), pp. 328–335 (cit. on p. 157).
- [Dad65] Luigi Dadda. “Some Schemes For Parallel Multipliers”. In: *Alta Frequenza* 45.5 (1965), pp. 349–356 (cit. on p. 176).
- [De +17] Davide De Caro, Ettore Napoli, Darjn Esposito, Gerardo Castellano, Nicola Petra, and Antonio GM Strollo. “Minimizing Coefficients Wordlength for Piecewise-Polynomial Hardware Function Evaluation With Exact or Faithful Rounding”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 64.5 (2017), pp. 1187–1200 (cit. on p. 794).
- [DIS13] Florent de Dinechin, Matei Istoan, and Guillaume Sergent. “Fixed-Point Trigonometric Functions on FPGAs”. In: *SIGARCH Computer Architecture News* 41.5 (2013), pp. 83–88 (cit. on p. 761).
- [DRC14] Theo A. Drane, Thomas M. Rose, and George A. Constantinides. “On the Systematic Creation of Faithfully Rounded Truncated Multipliers and Arrays”. In: *IEEE Transactions on Computers* 63.10 (2014), pp. 2513–2525 (cit. on p. 238).

- [EL04] Miloš D. Ercegovac and Tomás Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004
- [FF17] Christopher Fritz and Adly T. Fam. “Fast Binary Counters Based on Symmetric Stacking”. In: *Transactions on Very Large Scale Integration (VLSI) Systems* 25.10 (2017), pp. 2971–2975 (cit. on p. 180).
- [FS97] K. A. Feiste and Earl E. Swartzlander. “Merged Arithmetic Revisited”. In: *Workshop on Signal Processing Systems (SiPS)*. IEEE, 1997, pp. 212–221 (cit. on pp. 152, 157).
- [JPG06] Ghasssem Jaberipur, Behrooz Parhami, and Mohammad Ghodsi. “An Efficient Universal Addition Scheme for All Hybrid-Redundant Representations with Weighted Bit-Set Encoding”. In: *Journal of VLSI Signal Processing* 42 (2006), pp. 149–158 (cit. on p. 162).
- [JS94] Robert F. Jones and Earl E. Swartzlander. “Parallel Counter Implementation”. In: *Journal of VLSI signal processing systems for signal, image and video technology* 7.3 (1994), pp. 223–232 (cit. on p. 180).
- [KH11] Hou-Jen Ko and Shen-Fu Hsiao. “Design and Application of Faithfully Rounded and Truncated Multipliers With Combined Deletion, Reduction, Truncation, and Rounding”. In: *Transactions on Circuits and Systems II: Express Briefs* 58.5 (2011), pp. 304–308 (cit. on p. 238).
- [KK18] Martin Kumm and Johannes Kappauf. “Advanced Compressor Tree Synthesis for FPGAs”. In: *IEEE Transactions on Computers* 67.8 (2018), pp. 1078–1091 (cit. on pp. 183, 187, 189, 194, 195).
- [KS97] Eric J. King and Earl E. Swartzlander. “Data-Dependent Truncation Scheme for Parallel Multipliers”. In: *Asilomar Conference on Signals, Circuits and Systems*. Vol. 2. IEEE, 1997, pp. 1178–1182 (cit. on p. 238).
- [KZ14a] Martin Kumm and Peter Zipf. “Efficient High Speed Compression Trees on Xilinx FPGAs”. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*. 2014, pp. 171–182 (cit. on pp. 184, 185).
- [KZ14b] Martin Kumm and Peter Zipf. “Pipelined Compressor Tree Optimization Using Integer Linear Programming”. In: *International Conference on Field Programmable Logic and Application (FPL)*. IEEE, 2014, pp. 1–8 (cit. on p. 184).
- [LN96] Patrik Larsson and Chris J. Nicol. “Transition Reduction in Carry-Save Adder Trees”. In: *International Symposium on Low Power Electronics and Design*. IEEE, 1996, pp. 85–88 (cit. on p. 181).
- [Meo75] Angelo R. Meo. “Arithmetic Networks and Their Minimization Using a New Line of Elementary Units”. In: *IEEE Transactions on Computers* C-24.3 (1975), pp. 258–280 (cit. on p. 182).

- [MKM11] Taeko Matsunaga, Shinji Kimura, and Yusuke Matsunaga. "Power and Delay Aware Synthesis of Multi-Operand Adders Targeting LUT-Based FPGAs". In: *International Symposium on Low Power Electronics and Design (ISLPED)*. 2011, pp. 217–222 (cit. on p. [189](#)).
- [MKM13] Taeko Matsunaga, Shinji Kimura, and Yusuke Matsunaga. "An Exact Approach for GPC-Based Compressor Tree Synthesis". In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E96-A.12 (2013), pp. 2553–2560 (cit. on p. [189](#)).
- [MPS91] Mayur Mehta, Vijay Parmar, and Earl E. Swartzlander. "High-Speed Multiplier Design Using Multi-Input Counter and Compressor Circuits". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 1991, pp. 43–50 (cit. on pp. [180](#), [182](#)).
- [MTV06] Romain Michard, Arnaud Tisserand, and Nicolas Veyrat-Charvillon. "Carry Prediction and Selection for Truncated Multiplication". In: *Workshop on Signal Processing Systems (SiPS)*. IEEE, 2006, pp. 339–344 (cit. on p. [238](#)).
- [Par+11] Hadi Parandeh-Afshar, Arkosnato Neogy, Philip Brisk, and Paolo Ienne. "Compressor Tree Synthesis on Commercial High-Performance FPGAs". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 4.4 (2011), pp. 1–19 (cit. on pp. [157](#), [183](#), [187](#)).
- [Par10] Behrooz Parhami. *Computer Arithmetic, Algorithms and Hardware Designs*. 2nd ed. Oxford University Press, 2010 (cit. on pp. [23](#), [267](#), [287](#), [303](#)).
- [PBI08a] Hadi Parandeh-Afshar, Philip Brisk, and Paolo Ienne. "Efficient Synthesis of Compressor Trees on FPGAs". In: *Asia and South Pacific Design Automation Conference (ASPDAC)*. IEEE, 2008, pp. 138–143 (cit. on pp. [182](#), [183](#), [187](#)).
- [PBI08b] Hadi Parandeh-Afshar, Philip Brisk, and Paolo Ienne. "Improving Synthesis of Compressor Trees on FPGAs via Integer Linear Programming". In: *Design, Automation and Test in Europe (DATE)*. IEEE, 2008, pp. 1256–1261 (cit. on pp. [182](#), [189](#)).
- [Pet+10] Nicola Petra, Davide De Caro, Valeria Garofalo, Ettore Napoli, and Antonio G.M. Strollo. "Truncated Binary Multipliers With Variable Correction and Minimum Mean Square Error". In: *Transactions on Circuits and Systems I: Regular Papers* 57.6 (2010), pp. 1312–1325 (cit. on p. [238](#)).
- [PP01] Karuna Prasad and Keshab K. Parhi. "Low-Power 4-2 and 5-2 Compressors". In: *Asilomar Conference on Signals, Circuits and Systems*. IEEE, 2001, 129–133 vol.1 (cit. on pp. [181](#), [182](#)).
- [Pre17] Thomas B. Preußen. "Generic and Universal Parallel Matrix Summation with a Flexible Compression Goal for Xilinx FPGAs". In: *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–7 (cit. on p. [184](#)).

- [SDP11] Antonio G.M. Strollo, Davide De Caro, and Nicola Petra. "Elementary Functions Hardware Implementation Using Constrained Piecewise-Polynomial Approximations". In: *IEEE Transactions on Computers* 60.3 (2011), pp. 418–432 (cit. on p. 157).
- [SKG77] W. J. Stenzel, W. J. Kubitz, and G. H. Garcia. "A Compact High-Speed Parallel Multiplication Scheme". In: *IEEE Transactions on Computers* C-26.10 (1977), pp. 948–957 (cit. on p. 182).
- [SO96] Paul F. Stelling and Vojin G. Oklobdzija. "Design Strategies for Optimal Hybrid Final Adders in a Parallel Multiplier". In: *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology* 14.3 (1996), pp. 321–331 (cit. on p. 194).
- [SS93] Michael J. Schulte and Earl E. Swartzlander. "Truncated Multiplication with Correction Constant". In: *IEEE Workshop on VLSI Signal Processing*. 1993, pp. 388–396 (cit. on pp. 238, 241).
- [Swa04] Earl E. Swartzlander. "A Review of Large Parallel Counter Designs". In: *Annual Symposium on VLSI*. IEEE, 2004, pp. 89–98 (cit. on p. 180).
- [Swa73] Earl E. Swartzlander. "Parallel Counters". In: *IEEE Transactions on Computers* C-22.11 (1973), pp. 1021–1024 (cit. on p. 180).
- [Swa80] Earl E. Swartzlander. "Merged Arithmetic". In: *IEEE Transactions on Computers* C-29.10 (1980), pp. 946–950 (cit. on pp. 152, 157).
- [VBI08] Ajay K. Verma, Philip Brisk, and Paolo Ienne. "Data-Flow Transformations to Maximize the Use of Carry-Save Representation in Arithmetic Circuits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.10 (2008), pp. 1761–1774 (cit. on p. 157).
- [Vee+07] Sreehari Veeramachaneni, Avinash Lingamneni, M. Kirthi Krishna, and M. B. Srinivas. "Novel Architectures for Efficient (m, n) Parallel Counters". In: *17th ACM Great Lakes symposium on VLSI*. 2007 (cit. on p. 180).
- [Vol+19] Anastasia Volkova, Matei Istoan, Florent de Dinechin, and Thibault Hilaire. "Towards Hardware IIR Filters Computing Just Right: Direct Form I Case Study". In: *IEEE Transactions on Computers* 68.4 (2019) (cit. on p. 761).
- [Wei81] A. Weinberger. "4: 2 Carry-Save Adder Module". In: *IBM Technical Disclosure Bulletin* 23.8 (1981), pp. 3811–3814 (cit. on p. 181).
- [Yua+19] Yuelai Yuan, Le Tu, Kan Huang, Xiaoqiang Zhang, Tiejun Zhang, Dihu Chen, and Zixin Wang. "Area Optimized Synthesis of Compressor Trees on Xilinx FPGAs Using Generalized Parallel Counters". In: *IEEE Access* 7 (2019), pp. 134815–134827 (cit. on pp. 184, 194, 195).



8

CHAPTER 8

Fixed-Point Multiplication

Five, six, seven, eight! Fifty-six is seven times eight.

Anonymous

And God said to them, “Be fruitful and multiply”

Genesis 1:27–29

Multiplication is the next useful arithmetic building block after addition. In application-specific computing, hardware multipliers are often embedded in more complex operators. This chapter first reviews the many useful sizes and shapes that multipliers can take in a *computing just right* context and then addresses the construction of multipliers for ASICs and FPGAs.

Throughout this chapter, we consider the multiplication $X \times Y$, where X denotes the multiplier,¹ Y the multiplicand, and P the resulting product.

Section 8.1 discusses multiplier interfaces, with a special focus on non-standard ones (rectangular and truncated). It also discusses how multipliers impact error analysis and the other way around. This first section is necessary to understand many architectures in this book.

¹ The term “multiplier” denotes both an operator and one of its operands. This is unfortunate but mostly harmless, as the context usually avoids ambiguity.

The construction of multipliers is then addressed in the remaining sections. Section 8.2 gives a generic overview of the multiplier construction. The classic approaches used in the design of ASIC multipliers are reviewed in Sect. 8.3. Tiling approaches, which capture the decomposition of a multiplier into smaller ones, are addressed in Sect. 8.4. They are particularly relevant in modern heterogeneous FPGAs. The tiling methodology is further used to design truncated multipliers in Sect. 8.5 and to derive variants of Karatsuba’s algorithm in Sect. 8.6.

8.1 A Functional Point of View

This section reviews multipliers from a user point of view. We start with exact integer multipliers, both in the unsigned and signed case. We then generalize to fixed-point multipliers, which may be inexact due to rounding and/or truncation. Finally, we discuss error analysis when multipliers are involved, from which we draw general guidelines on the use of multipliers in application-specific arithmetic.

8.1.1 Integer Multipliers

8.1.1.1 Exact Multiplication of Unsigned Integers

Let us first formalize the multiplication of two unsigned integer numbers, X on w_X bits and Y on w_Y bits. The product is an integer, and its maximum value is $(2^{w_X} - 1) \times (2^{w_Y} - 1) < 2^{w_X + w_Y}$: it fits on a $(w_X + w_Y)$ -bit unsigned integer.

Examples of binary integer multiplications computed by the schoolbook method are given in Fig. 8.1. This schoolbook method involves the computation of $w_X \times w_Y$ partial products, arranged in a parallelogram as illustrated by Fig. 8.1a. This alignment is the proper point of view when building a multiplier. However, the reader should remember that multiplication is commutative: Fig. 8.1b represents the same multiplication. Although both figures look different, they lead to the exact same dot diagram and introduced in Chap. 7 and given in Fig. 8.1c (remember that the order in a column of a dot diagram does not matter).

The symmetry due to commutativity is better captured when the $w_X \times w_Y$ bit array of the multiplier is represented as a rectangle, as in Fig. 8.2. For this reason, multipliers for which $w_X \neq w_Y$ are often called *rectangular multipliers* in the literature. For instance, the DSP blocks (see Sect. 4.2) of some AMD FPGAs are based on rectangular, e.g., 18×27 bits, multipliers. This

(a) $X \times Y$

1	1	0	0	1	0	1	1
\times	1	0	0	1	0		
<hr/>							
0	0	0	0	0	0	0	0
1	1	0	0	1	0	1	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	0	0	1	0	1	1
<hr/>							
0	1	1	1	0	0	1	0

(b) $Y \times X$

1	0	0	1	0			
\times	1	1	0	0	1	0	1
<hr/>							
1	0	0	1	0	1	0	0
0	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0
1	0	0	1	0	1	0	0
<hr/>							
0	1	1	1	0	0	0	1

(c) Dot diagram (bit heap) in both cases

Fig. 8.1 Schoolbook view of binary integer multiplication $X \times Y$ when $w_X \neq w_Y$.

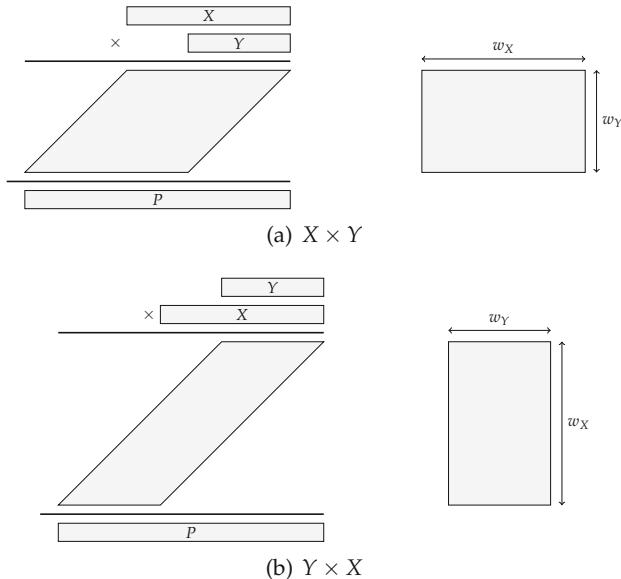


Fig. 8.2 Abstract representations of a rectangular binary integer multiplication: the parallelogram on the left is a sheared view of the rectangle on the right.

classic formulation (“ $w_X \times w_Y$ multiplier”) actually describes the shape of the rectangular bit array.

8.1.1.2 Exact Multiplication of Signed Integers

When both X and Y are signed numbers in two’s complement, one subtlety arises due to the asymmetry of the set of representable numbers.

The largest representable value for X is $2^{w_X-1} - 1$, its smallest value is -2^{w_X-1} , and similar formulas hold for Y . Now, the largest possible value of the product is the product of the two extremal negative values: $(-2^{w_X-1}) \times (-2^{w_Y-1}) = 2^{w_X+w_Y-2}$. The smallest possible value is either $-2^{w_X-1}(2^{w_Y-1} - 1)$ or $-2^{w_Y-1}(2^{w_X-1} - 1)$.

Again the product fits on a signed integer of $w_X + w_Y$ bits, but the subtlety is that half of the range offered by this format is unused: all the possible values of P would fit in a signed number of $w_X + w_Y - 1$ bits, except the single extremal value $2^{w_X+w_Y-2}$.

The fact that the integer range for the product of two signed integers is almost half that of the product of two unsigned integers can be intuitively explained by the fact that there are always two combinations of the input numbers, $(-X) \times Y$ and $X \times (-Y)$, that lead to the same product.

We will come back to this issue when dealing with arbitrary fixed-point numbers.

8.1.1.3 Exact Multiplication of an Unsigned Integer by a Signed Integer

The product of an unsigned w_X -bit integer by a signed w_Y -bit integer is obviously a signed integer. We leave it as an exercise to the reader to check that it again fits in a $(w_X + w_Y)$ -bit signed integer.

8.1.2 Fixed-Point Numbers

In the present section, the numbers X , Y , and P are, in all generality, fixed-point numbers. We note, following the standard notations introduced in Chap. 2,

- m_X the most significant bit (MSB) of X ,
- ℓ_X its least significant bit (LSB),
- and $w_X = m_X - \ell_X + 1$ its size in bits,

and similarly for the other input Y and for the product P . The corresponding fixed-point formats may be unsigned, i.e., $\text{ufix}(m_X, \ell_X)$, or signed, i.e., $\text{sfix}(m_X, \ell_X)$. Table 8.1 recalls the extremal values of a fixed-point format.

The parameters of X and its signedness are not necessary identical to those of Y . The parameters of the product P may be specified independently

Table 8.1 Extremal values of fixed-point numbers.

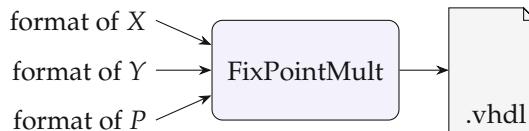
Format	$\text{ufix}(m, \ell)$	$\text{sfix}(m, \ell)$
Max value	$2^{m+1} - 2^\ell$	$2^m - 2^\ell$
Min value	0	-2^m
Max absolute value	$2^{m+1} - 2^\ell$	2^m
Min non-zero value	2^ℓ	2^ℓ

as well. The interface to a generic multiplier generator is thus the one shown in Fig. 8.3.

Depending on the respective values of these parameters,

- a multiplier may never overflow, or may risk overflow;
- a multiplier may always be exact, or may require rounding or truncation.

This may appear to the reader as a case of over-parameterization, and indeed not all these variations will be used in this book. However, to decide convincingly which of these multiplier variants is useful in which context of application-specific arithmetic, we follow the methodology introduced in Chap. 3: first over-parameterize the problem with serenity, then perform a parametric error analysis, and finally use this analysis to discuss the optimal value of the parameters.

**Fig. 8.3** Interface to a generator of fixed-point multipliers.

8.1.2.1 Exact Multiplication of Fixed-Point Numbers

Remembering that a fixed-point number, be it $\text{ufix}(m, \ell)$ or $\text{sfix}(m, \ell)$, is an integer scaled by 2^ℓ , we can easily derive from the analysis of integer multiplication the following properties:

- The size of the smallest format ensuring that all products can be represented exactly is always $w_P = w_X + w_Y$.
- The LSB of the exact product always has the binary weight $\ell_P = \ell_X + \ell_Y$. A multiplication targeting an ℓ_P larger than $\ell_X + \ell_Y$ means that the product is rounded or truncated somehow.
- The MSB of the exact product always has the binary weight $m_P = m_X + m_Y + 1$.

- In the signed times signed case, the most significant bit is only used by one single combination of the inputs, the product $(-2^{m_X}) \times (-2^{m_Y})$.

The schoolbook algorithm can also trivially be extended to fixed point, as illustrated in Fig. 8.4. However, it becomes a bit more tricky with signed numbers (sign extension must be used). In particular, it is extremely difficult to derive from such a figure the possible saving of the most significant bit (MSB) that we now discuss.

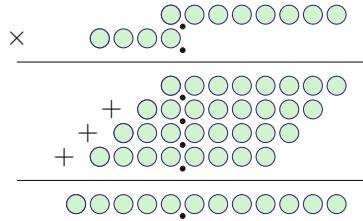


Fig. 8.4 Bit-array representation of an exact multiplier of $\text{ufix}(0, -7)$ times $\text{ufix}(3, 0)$.

8.1.2.2 The MSB Issue in Signed Fixed-Point Multiplication

The fact that almost half the output range of a signed multiplier is unused is a first case for computing just right.

Let us first illustrate for simplicity on a practically relevant case: the product of two w -bit signed numbers in $[-1, 1]$. Assume the input format for both inputs is $\text{sfix}(0, \ell)$ with $\ell = -w + 1$. The exact product fits in the format $\text{sfix}(1, 2\ell)$: its sign bit has weight 2^1 . However, the product could almost fit a $\text{sfix}(0, 2\ell)$ format: the only case that requires the extra MSB is the product $(-1) \times (-1) = +1$ which is not representable in $\text{sfix}(0, 2\ell)$.

When designing circuits that compute just right, it is an issue to have a bit that is almost never useful. Therefore, we will look for ways to avoid it. The proper solution will be application-specific. Here are a few examples.

- In many applications, it can be part of the specification that the signal should belong to $(-1, 1)$, sacrificing the possible input value -1 .
- The computation of many digital filters is based on sums of products by constant coefficients. When designing such filters, it is usually possible to ensure that none of these coefficients reaches the problematic minimal negative value 2^{m_Y} . Then, even when the input signal may reach -2^{m_X} , the output will fit in the $\text{sfix}(0, \ell_X + \ell_Y)$ format.
- In Chap. 20, we define a slightly scaled sine function: $(1 - 2^{-\ell_R}) \sin(\pi x)$, where ℓ_R is the LSB of the output. This scaled sine function is specified to never reach -1 . This ensures that any multiplier that inputs the result of this function can save the almost useless MSB.

In summary, when multiplying two signed numbers, there is an opportunity to drop the MSB of the output. Note that it does not entail much hardware saving in the multiplier itself: the main saving will be in subsequent operations, which can consume one bit less.

8.1.2.3 Rounded and Truncated Fixed-Point Multipliers

Using only exact multipliers in an application would entail that the sizes of the internal datapaths increase along with the computation. This is why the results of these multipliers must be truncated or rounded, which we address now.

The fully parametric multiplier generator (whose interface is shown on Fig. 8.3) inputs three completely independent fixed-point formats, for X , Y , and P . If $\ell_P \leq \ell_X + \ell_Y$, then the result is exact. If $\ell_P > \ell_X + \ell_Y$, then the product must be rounded somehow. We denote $u = 2^{\ell_P}$ the unit in the last place (ulp) of the result. As discussed in Chap. 3, we have two options: rounding to the nearest (with an error bounded by $u/2$) and last-bit accuracy (with an error bounded by u).

As Sect. 8.5 will show, for an accuracy objective of 2^k , a multiplier last-bit accurate to $\ell_P = k$ is in general much cheaper to build than a multiplier rounded to the nearest to $\ell_P = k + 1$. The choice made in FloPoCo is therefore to provide last-bit accurate multipliers.

8.1.3 Error Analysis Involving Multipliers

Let us now address the issue of error propagation in a multiplier, for the general situation where both inputs may themselves be inexact:

$$X = X_0 + \delta_X \quad (8.1)$$

$$Y = Y_0 + \delta_Y . \quad (8.2)$$

Here X_0 and Y_0 are reference values for the inputs X and Y , and δ_X and δ_Y capture the accumulated approximation and rounding errors of computing these inputs so far.

The product, in the general case, can be expressed as

$$P = X \times Y + \delta_{\text{mult}} \quad (8.3)$$

$$= (X_0 + \delta_X) \times (Y_0 + \delta_Y) + \delta_{\text{mult}} \quad (8.4)$$

$$= X_0 Y_0 + \underbrace{\delta_X Y_0 + \delta_Y X_0 + \delta_X \delta_Y}_{=\delta_P} + \delta_{\text{mult}} . \quad (8.5)$$

An upper bound on the overall error on the product can be derived by the triangle inequality:

$$|\delta_P| \leq |\delta_X Y_0| + |\delta_Y X_0| + |\delta_X \delta_Y| + |\delta_{\text{mult}}| . \quad (8.6)$$

Let us note \overline{X}_0 and \overline{Y}_0 the maximum absolute values that X_0 and Y_0 may take. By default, these will be deduced from the format and taken from Table 8.1. However, it is often the case that the application comes with slightly smaller bounds, which can make a difference. Of course, if the bound is less than half the bound of Table 8.1, the corresponding input format should be reduced. The bound on the overall error of the product becomes

$$\bar{\delta}_P = \overline{Y}_0 \cdot \bar{\delta}_X + \overline{X}_0 \cdot \bar{\delta}_Y + \bar{\delta}_X \cdot \bar{\delta}_Y + \bar{\delta}_{\text{mult}} . \quad (8.7)$$

Here, we have a second-order error term $\bar{\delta}_X \cdot \bar{\delta}_Y$ which will usually be negligible. However, the three other error terms may all be important in an architecture computing just right.

- $\bar{\delta}_{\text{mult}}$ is the term that corresponds to the error introduced by the multiplier. It will usually be equal to $2^{-\ell_P}$, but it may be null if an exact multiplier is used.
- $\overline{Y}_0 \cdot \bar{\delta}_X$ and $\overline{X}_0 \cdot \bar{\delta}_Y$ are the terms corresponding to the amplification by the multiplier of the errors on its input. They will remain, even if an exact multiplier is used!

Now, we may use this error analysis to infer architectural guidelines with respect to truncated multipliers. Using the values of \overline{X}_0 and \overline{Y}_0 from Table 8.1 in the signed case (the other cases are similar), and assuming that X and Y are relatively accurate, i.e., $\bar{\delta}_X \approx 2^{\ell_X}$ and $\bar{\delta}_Y \approx 2^{\ell_Y}$, we find that

$$\overline{Y}_0 \cdot \bar{\delta}_X \approx 2^{m_Y} 2^{\ell_X} = 2^{m_Y + \ell_X} \quad (8.8)$$

$$\overline{X}_0 \cdot \bar{\delta}_Y \approx 2^{m_X} 2^{\ell_Y} = 2^{m_X + \ell_Y} . \quad (8.9)$$

If one of these terms is much smaller than the other, it means that the architecture is computing uselessly accurately somewhere. We therefore try to have $\overline{Y}_0 \cdot \bar{\delta}_X \approx \overline{X}_0 \cdot \bar{\delta}_Y$, which is achieved when $2^{m_Y + \ell_X} \approx 2^{m_X + \ell_Y}$, which can be rewritten $m_Y + \ell_X \approx m_X + \ell_Y$ or in other terms $w_Y \approx w_X$.

The conclusion of this analysis is therefore that **architectures computing just right should favor truncated square multipliers**. Besides, we also want to balance the term $\bar{\delta}_{\text{mult}} = 2^{-\ell_P}$ with the two others (again to make sure that no part of the architecture is ridiculously accurate compared to other parts). For this we should have

$$\ell_P \approx m_Y + \ell_X \approx m_X + \ell_Y . \quad (8.10)$$

This guideline will be a good rule of thumb in the early architectural design phase. Then, as the architecture is being detailed, the previous error analysis can be refined accordingly. For instance, we assumed $\bar{\delta}_X \approx 2^{\ell_X}$, but in deep computation pipelines, we may have $\bar{\delta}_X = K \cdot 2^{\ell_X}$ with some $K > 1$ capturing the accumulation of all the errors in the computation leading to X. This is easy to take into account by instantiating (8.7) properly.

8.1.4 Summary: Multipliers for Computing Just Right

Here are a few take-away messages from the previous analysis.

Exact multipliers, including rectangular ones, are useful in practice.

They may be available as a building block in some FPGA architectures. They can be useful for building larger multipliers, using techniques that will be introduced in Sect. 8.4.2. They can be useful to implement cryptographic primitives or fancy arithmetic such as Residue Number System (RNS) arithmetic [Kor02].

Truncated square multipliers are very useful in practice.

In a general error analysis where none of the inputs is exact, there is a soft spot in terms of multiplier area when $w_X \approx w_Y$ for a result that is last-bit accurate to ℓ_P , and ℓ_P should fulfill (8.10). The construction of truncated multipliers is discussed in Sect. 8.5.

Truncated rectangular multipliers may be useful in practice, but rarely.

For a truncated rectangular multiplier to make sense, one of the inputs must be exact. For instance, if X is exact, or in other terms $\bar{\delta}_X = 0$, then (8.10) is simplified in $\ell_P \approx m_X + \ell_Y$, and we no longer have the constraint that w_X should be close to w_Y . Let us take a practical example. The obvious way to compute the logarithm of a floating-point number $2^E \cdot M$ is

$$\log(2^E \cdot M) = E \times \log(2) + \log(M) . \quad (8.11)$$

In the product $E \times \log(2)$, the constant $\log(2)$ will be a large fixed-point number, while E is an exact integer, with no δ associated to it. It is a case for a truncated rectangular multiplier.

In Chap. 22 we again have to compute a product $E \times \log(2)$, with a new subtlety: there E is a tentative exponent; it is not, properly speaking, exact. Indeed, a later step will check if this tentative value does lead to an overflow and update the value of E accordingly. However, the computation leading to this potential overflow speculates on an exact value.

Any case of truncated rectangular multiplier where none of the inputs is exact should raise suspicion that some hardware could be saved in the multiplier by truncating both inputs to implement (8.10) and this before the multiplication.

Multipliers whose MSB is smaller than that of an exact multiplier (i.e., such that $m_P < m_X + m_Y$ in the signed case or $m_P < m_X + m_Y + 1$ in all the other cases) are of little use in practice.

Such multipliers can silently overflow, “silently” meaning that the result of an overflow cannot be detected as such: this is usually catastrophic. Potential overflows are acceptable only if they are detected and reported somehow to the application. But to detect overflows, the simplest architecture is a non-overflowing multiplier and then some overflow detection logic on its leading bits.

Interestingly, the multiplication $E \times \log(2)$ in Chap. 22 is also one of the very few occurrences in this book where we have $m_P < m_X + m_Y + 1$. Indeed, the purpose there is to compute a reduced argument Y as

$$Y = X - E \times \log(2) \quad (8.12)$$

and the integer E has been itself computed to make sure the subsequent subtraction $X - E \times \log(2)$ cancels the leading bits of X . In this case, since it is known by construction that these leading bits will be zero, there is no need to dedicate hardware to compute these zeroes: it is possible to save the leading bits of the product and also the leading bits of the subtraction.

The same trick may be used in the range reduction for the logarithm function [DDP07]. It is also at the core of the Payne and Hanek argument reduction for trigonometric functions on very large arguments [Mul16] which has been implemented in software [Ng92] and in hardware [DD07a].

As far as we know, any other case of multiplier for which $m_P < m_X + m_Y + 1$ should raise a suspicion that something catastrophic is waiting to happen.

8.2 Overview of Multiplier Construction

The remainder of this chapter discusses the construction of multipliers.

As a fixed-point number is an integer scaled by some implicit 2^ℓ , an exact fixed-point multiplier can be viewed as an exact integer multiplier whose output is scaled by $2^{\ell_X + \ell_Y}$. This scaling factor is implicit; therefore for the purpose of multiplier construction, we may focus on integer multipliers. This will be simpler since an integer format has only one parameter, its bit size w . A second parameter (for the output) will only be introduced when constructing truncated and rounded multipliers.

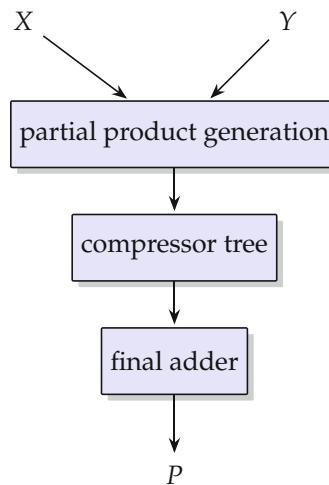


Fig. 8.5 Data-flow of a parallel multiplier.

It is possible to build sequential multipliers that iterate additions, but we mostly focus here on *parallel* multipliers which offer better throughput. The computation flow of a parallel multiplier is illustrated in Fig. 8.5. First, partial products are computed from the inputs. In the schoolbook method (Fig. 8.1), each partial product bit is obtained by ANDing each bit of X with each bit of Y , but more efficient multipliers may be designed by considering a higher radix for X and Y : this will be detailed in Sect. 8.3, which also addresses sign management. Next, the weighted partial products are summed up in a compressor tree. Finally, the carry-save form of the compressed result has to be added. These two steps were already addressed in detail in Chap. 7.

In terms of delay, partial product generation is in constant time since all the partial products can be computed in parallel. Using techniques shown in Chaps. 7 and 5, the compressor tree and the final adder can both be computed in $\mathcal{O}(\log w)$ time (using Landau's “big-O” notation) and $w =$

$\max(w_X, w_Y)$. The flow in Fig. 8.5 therefore enables the construction of multipliers in $\mathcal{O}(\log w)$ as well.

Conversely, some multiplier schemes combine the partial product generation with their summation [EL04; KAZ15; Wal16]. These array-like multipliers offer a regular layout but are typically slower than multipliers based on compressor trees, with a delay in $\mathcal{O}(w)$.

Multipliers on FPGAs essentially follow the same flow in Fig. 8.5. However, they offer a range of heterogeneous resources that can be used to implement multipliers: look-up tables (LUTs), carry chains, and digital signal processing (DSP) blocks. The efficient use of these resources can be captured as a geometric tiling problem, which can be solved thanks to integer linear programming (ILP). This is the subject of Sect. 8.4.

The construction of truncated multipliers also follows the same flow, but starting with a smaller subset of partial products. This is studied in Sect. 8.5.

Finally, Sect. 8.6 studies the Karatsuba method that reduces the hardware cost of multipliers at the expense of delay.

8.3 Partial Product Generation and Sign Management

The simplest case of unsigned multiplication using the schoolbook method (Fig. 8.1) corresponds to a binary (radix-2) partial product generation. As already introduced as motivation for the bit heaps in Sect. 7.1.1, it can be written as

$$P = X \times Y = \left(\sum_{i=0}^{w_X-1} 2^i x_i \right) \times Y = X \times \left(\sum_{j=0}^{w_Y-1} 2^j y_j \right) = \quad (8.13)$$

$$= \sum_{i=0}^{w_X-1} \sum_{j=0}^{w_Y-1} 2^{i+j} x_i y_j \quad (8.14)$$

Here, each partial product $x_i y_j$ is either 0 or 1 and can be computed by a simple AND gate. For the example of $w_X = w_Y = 4$ bits, the corresponding partial products and their alignment are shown in Fig. 8.6.

Remark that radix-2 has the very nice property that the product of two digits is written as one digit only. For any radix higher than 2 (e.g., radix-10), the product of two digits must be written as two digits.

As Fig. 8.1 shows, the partial products are organized either as w_X rows of w_Y bits each or as w_Y rows of w_X bits each. In both cases we obtain $w_Y \times w_X$ bits that have to be compressed. Furthermore, even when $w_Y \neq w_X$, expanding X or expanding Y leads to the same shape for the bit heap.

Without loss of generality, we always expand X in the following, and we note $w = w_X$ to lighten notations.

$$\begin{array}{r}
 & x_0y_3 & x_0y_2 & x_0y_1 & x_0y_0 \\
 + & x_1y_3 & x_1y_2 & x_1y_1 & x_1y_0 \\
 + & x_2y_3 & x_2y_2 & x_2y_1 & x_2y_0 \\
 + & x_3y_3 & x_3y_2 & x_3y_1 & x_3y_0 \\
 \hline
 = & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
 \end{array}$$

Fig. 8.6 Partial product alignment of an unsigned 4×4 bit multiplier.

8.3.1 Signed Multiplication in Radix-2

In the signed case where both inputs are represented in two's complement, we get the following:

$$X \times Y = \left(-2^{w-1}x_{w-1} + \sum_{i=0}^{w-2} 2^i x_i \right) \times Y \quad (8.15)$$

$$= -2^{w-1}x_{w-1}Y + \sum_{i=0}^{w-2} 2^i x_i Y \quad (8.16)$$

As Y is now a two's complement number, this leads to the following changes in the partial product generation:

1. Each $2^i x_i Y$ must be sign-extended to the output word size.
2. One partial product line ($x_{w-1}Y$) has to be subtracted. This is done by adding its two's complement. Hence, these bits must be negated, and a 1 must be added at the LSB position of this line.

The initial partial products and their alignment are shown for a signed 4×4 bit multiplier in Fig. 8.7a. We can then apply the sign extension reduction technique introduced in Sect. 7.2.2: Fig. 8.7b shows the partial product array after replacing the sign extensions with constant vectors. Finally, Fig. 8.7c shows the result of adding up all the constant bits (including the one for negating $x_{w-1}Y$). Observe the symmetry of the pattern of bits that must be negated, reflecting that X and Y can be exchanged.

Altogether, the overhead of sign management in a two's complement multiplier reduces to a few constant bits in the partial product array.

8.3.2 Radix-4 Multiplication Using Booth Encoding

In radix-2 there is a total of $w_Y \times w_X$ partial product bits, plus a line of constant bits in the signed case. Let us now see how this number can almost be halved by considering for X a high-radix binary representation (as introduced in Sect. 2.3). For this, X is split into chunks of 2 bits, denoted X_i , such

$$\begin{array}{r}
 \begin{array}{cccccccccc}
 x_0y_3 & x_0y_3 & x_0y_3 & x_0y_3 & x_0y_3 & x_0y_2 & x_0y_1 & x_0y_0 \\
 + & x_1y_3 & x_1y_3 & x_1y_3 & x_1y_3 & x_1y_2 & x_1y_1 & x_1y_0 \\
 + & x_2y_3 & x_2y_3 & x_2y_3 & x_2y_2 & x_2y_1 & x_2y_0 \\
 + & \overline{x_3y_3} & \overline{x_3y_3} & \overline{x_3y_2} & \overline{x_3y_1} & \overline{x_3y_0} \\
 + & & & & & 1
 \end{array} \\
 = p_7 \quad p_6 \quad p_5 \quad p_4 \quad p_3 \quad p_2 \quad p_1 \quad p_0
 \end{array}$$

(a) Initial partial products

$$\begin{array}{r}
 \begin{array}{cccccccccc}
 & & & & \overline{x_0y_3} & x_0y_2 & x_0y_1 & x_0y_0 \\
 + & 1 & 1 & 1 & 1 & 1 & & & \\
 + & & & & \overline{x_1y_3} & x_1y_2 & x_1y_1 & x_1y_0 \\
 + & 1 & 1 & 1 & 1 & & & & \\
 + & & & & \overline{x_2y_3} & x_2y_2 & x_2y_1 & x_2y_0 \\
 + & 1 & 1 & 1 & & & & & \\
 + & \overline{x_3y_3} & \overline{x_3y_3} & \overline{x_3y_2} & \overline{x_3y_1} & \overline{x_3y_0} \\
 + & & & & & 1
 \end{array} \\
 = p_7 \quad p_6 \quad p_5 \quad p_4 \quad p_3 \quad p_2 \quad p_1 \quad p_0
 \end{array}$$

(b) Intermediate step after sign extension replacement

$$\begin{array}{r}
 \begin{array}{cccccccccc}
 & & & & 1 & \overline{x_0y_3} & x_0y_2 & x_0y_1 & x_0y_0 \\
 + & & & & \overline{x_1y_3} & x_1y_2 & x_1y_1 & x_1y_0 \\
 + & 1 & 1 & \overline{x_2y_3} & x_2y_2 & x_2y_1 & x_2y_0 \\
 + & \overline{x_3y_3} & \overline{x_3y_3} & \overline{x_3y_2} & \overline{x_3y_1} & \overline{x_3y_0} \\
 + & & & & & 1
 \end{array} \\
 = p_7 \quad p_6 \quad p_5 \quad p_4 \quad p_3 \quad p_2 \quad p_1 \quad p_0
 \end{array}$$

(c) Optimized partial products after compressing the constant ones

Fig. 8.7 Partial product alignment of a signed 4×4 bit multiplier.

that

$$X = \sum_{i=0}^{w/2-1} 4^i X_i \quad \text{with} \quad X_i \in \{0, 1, 2, 3\}. \quad (8.17)$$

We assume here that $w = w_X$ is divisible by 2; otherwise X has to be extended by one zero bit. Putting this into the multiplication yields

$$P = X \times Y = \left(\sum_{i=0}^{w/2-1} 4^i X_i \right) \times Y. \quad (8.18)$$

The number of terms is halved (thus also halving the size of the compressor tree that inputs them). However, each term is now the product of Y by a digit $X_i \in \{0, 1, 2, 3\}$. The multiplication by $X_i = 0$ and $X_i = 1$ is as trivial as for radix-2, the multiplication by $X_i = 2$ can be simply realized by a bit-shift of Y , but a multiplication by $X_i = 3$ would require an extra addition

to compute $3Y = 2Y + Y$. The advantage of halving the number of partial products is partially annihilated by the additional complexity of an extra addition, which furthermore lies in the critical path.

A solution to avoid this extra addition is a recoding scheme initially proposed by Booth [Boo51] and modified for the present context and hence usually referred to as *modified Booth recoding*. In short, this technique consists in rewriting X as

$$X = \sum_i 4^i Z_i \quad \text{with} \quad Z_i \in \{-2, -1, 0, 1, 2\}. \quad (8.19)$$

This is still a radix-4 representation (a number is a sum of digits Z_i , each weighted by 4^i), but a redundant number system (see Sect. 2.4, p. 46) with a digit set including the negative digits $\{-2, -1, 0, 1, 2\}$ instead of $\{0, 1, 2, 3\}$. Now, the product becomes

$$X \times Y = \sum_i 4^i Z_i Y \quad (8.20)$$

and all the $Z_i Y$ can be computed as a simple bit-shift of Y . Some of these products are now negative, but we know how to manage negative terms in a bit heap.

Let us now show how to perform the rewriting of X in (low) constant time and at a small hardware cost. Modified Booth recoding is based on the simple identity $x = 2x - x$, which we first apply to rewrite all the binary digits of X with odd weight:

$$\begin{aligned} X &= \dots + x_5 2^5 + x_4 2^4 + x_3 2^3 + x_2 2^2 + x_1 2^1 + x_0 \\ &= \dots + (2x_5 - x_5) 2^5 + x_4 2^4 + (2x_3 - x_3) 2^3 + x_2 2^2 + (2x_1 - x_1) 2^1 + x_0 . \end{aligned}$$

The bits can then be reassigned to the new radix-4 digits Z_j :

$$X = \dots + 2x_5 2^5 + 2^4 \underbrace{(-2x_5 + x_4 + x_3)}_{=Z_2} + 2^2 \underbrace{(-2x_3 + x_2 + x_1)}_{=Z_1} + \underbrace{(-2x_1 + x_0)}_{=Z_0} \quad (8.21)$$

To simplify (8.21) so that each digit Z_j depends on exactly three consecutive bits of X , we extend X with a zero bits on the right ($x_{-1} = 0$). We also extend it with one or two zero bits on the left: if w is even, we add $x_w = 0$ and $x_{w+1} = 0$ (e. g., in (8.21) with $w = 4$, adding $x_4 = 0$ and $x_5 = 0$ completes the definition of Z_2); if w is odd, we add $x_w = 0$ (e. g., in (8.21) with $w = 3$, adding $x_3 = 0$ completes the definition of Z_1).

With this zero padding (which will entail no additional hardware cost as these extra bits will be optimized out by the logic synthesis tools), we obtain

$$X = \sum_{j=0}^{m-1} Z_j 2^{2j} \quad \text{with} \quad Z_j = -2x_{2j+1} + x_{2j} + x_{2j-1} \quad \text{and} \quad m = \left\lfloor \frac{w}{2} + 1 \right\rfloor. \quad (8.22)$$

The formula $Z_j = -2x_{2j+1} + x_{2j} + x_{2j-1}$ indeed ensures $Z_j \in \{-2, -1, 0, 1, 2\}$, thus avoiding the unwanted multiplication by 3. It is also important to point out that the Z_j can be computed out of the x_i fully in parallel, hence in constant time. Table 8.2 shows the truth table of all the possible digit values.

Take, for example, $X = 123_{10}$. Its (zero-padded) binary representation is

$$X = 123_{10} = \overline{01111011.0}_2. \quad \begin{matrix} \\ \overline{Z_3} \overline{Z_2} \overline{Z_1} \overline{Z_0} \end{matrix}$$

Booth recoding leads to the digits

$Z_0 = -1$	for input vector (11.0)
$Z_1 = -1$	for input vector (101)
$Z_2 = 0$	for input vector (111)
$Z_3 = 2$	for input vector (011)

and the value of this recoding is $X = (-1) + 2^2 \cdot (-1) + 2^4 \cdot 0 + 2^6 \cdot 2 = 123_{10}$.

8.3.2.1 Technical Realization of Radix-4 Booth Encoding

In practice, it is possible to fuse the recoding and the multiplication $Z_j Y$. For this purpose, a simple encoding of the five possible values of a digit Z_j is as the following three bits [EL04]:

- $\text{cpl}_j = 1$ if Z_j is negative or zero,
- $\text{one}_j = 1$ if Z_j is either 1 or -1 ,
- $\text{two}_j = 1$ if Z_j is either 2 or -2 ,

Table 8.2 Truth table of the Booth encoded digit values Z_j .

x_{2j+1}	x_{2j}	x_{2j-1}	$Z_j = -2x_{2j+1} + x_{2j} + x_{2j-1}$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

which can be computed by the following Boolean equations:

$$\text{cpl}_j = x_{2j+1}$$

$$\text{one}_j = x_{2j} \oplus x_{2j-1}$$

$$\text{two}_j = \overline{x_{2j+1}} x_{2j} x_{2j-1} \vee x_{2j+1} \overline{x_{2j}} x_{2j-1}.$$

Thus, the generation of these signals requires a few simple gates, replicated m times. The computation of a partial product line $Z_j Y$ using this encoding of Z_j is illustrated in Fig. 8.8. A first stage selects either Y or $2Y$ (obtained from a left shift). A second stage inverts it by an XOR in case $Z_j \leq 0$ (i.e., $\text{cpl}_j = 1$). To complete the negation in this case, cpl_j must also be added at the LSB position of $4^j Z_j Y$.

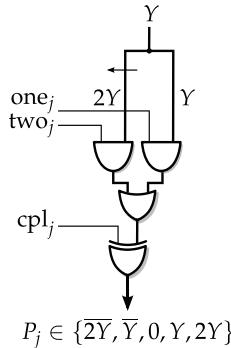


Fig. 8.8 Partial product generation using radix-4 Booth recoding.

Each gate in Fig. 8.8 actually represents w_Y Boolean gates, and this figure is itself replicated m times. As $m \approx w_X/2$, the two AND gates on Fig. 8.8 replace those computing the binary partial products in a non-Booth multiplier: the overhead of Booth encoding is really only the $m \times w_Y$ OR and XOR gates, as well as the computation of cpl_j , one_j , and two_j . For this price, the size of the bit array is almost divided by two.

In detail, we must distinguish between the unsigned and signed cases.

8.3.2.2 Unsigned Radix-4 Booth Partial Product Generation

As each Z_j can be negative, each $Z_j Y$ can be negative as well. Hence, we need to sign-extend all the $Z_j Y$ even for an unsigned multiplication.

Figure 8.9a shows the initial partial product alignment for an unsigned 4×4 bit Booth multiplier without further optimization, where p_{ji} denotes the i -th bit of the partial product line $P_j = X_j \times Y$. For our small (and academic) example of four bits, we need three Booth-encoded digits to represent X .

Each P_j consists of $w_Y + 1$ bits before sign extension (due to the possible multiplication by 2) except for the most significant partial product, here P_2 : As the two leading digits of X are the extension zeros, Z_2 can be only 0 or 1 (see Table 8.2). Therefore, no word size extension is necessary for P_2 . For the same reason, cpl_2 is always zero and does not have to be added.

The step after performing the sign extension transformations is shown in Fig. 8.9b. We can at least replace four digits by three constant ones, but we increased the column height from three to four. Hence, the algebraic transform for $x + 1$ (see Fig. 7.14, p. 164) can be applied two times to get a height reduced version as shown in Fig. 8.9c. This has at most three bits per column. On this very small example, Booth recoding only reduces the height of the bit array by one bit, but on large multipliers the size of the bit array is almost divided by 2.

$$\begin{array}{ccccccccc}
 p_{04} & p_{04} & p_{04} & p_{04} & p_{03} & p_{02} & p_{01} & p_{00} \\
 + & & & & & & & \text{cpl}_0 \\
 + p_{14} & p_{14} & p_{13} & p_{12} & p_{11} & p_{10} & & \\
 + & & & & & & \text{cpl}_1 \\
 + p_{23} & p_{22} & p_{21} & p_{20} & & & & \\
 \hline
 = p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
 \end{array}$$

(a) *Initial partial products*

$$\begin{array}{ccccccccc}
 & & \overline{p_{04}} & p_{03} & p_{02} & p_{01} & p_{00} \\
 + & \overline{p_{14}} & p_{13} & p_{12} & p_{11} & p_{10} \\
 + p_{23} & p_{22} & p_{21} & p_{20} & & \text{cpl}_1 & \text{cpl}_0 \\
 + 1 & & 1 & 1 & & & \\
 \hline
 = & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
 \end{array}$$

(b) After sign extension replacement

$$\begin{aligned}
 & + 1 \frac{\overline{p_{04}}}{p_{14}} p_{04} p_{04} p_{03} p_{02} p_{01} p_{00} \\
 & + p_{23} p_{22} p_{21} p_{20} \text{cpl}_1 \text{cpl}_0 \\
 = & p_7 \quad p_6 \quad p_5 \quad p_4 \quad p_3 \quad p_2 \quad p_1 \quad p_0
 \end{aligned}$$

(c) Height reduced partial products using algebraic optimization

Fig. 8.9 Partial product alignment of an unsigned 4×4 bit Booth multiplier.

8.3.2.3 Signed Radix-4 Booth Partial Product Generation

In the signed Booth multiplication, we first have to deal with the Booth encoding of a two's complement number, which is slightly different for the MSB part. We have two cases here:

If w is even, one Booth digit covers exactly the three MSBs of X :

$$\frac{x_{w-1}x_{w-2}x_{w-3}x_{w-4}x_{w-5}\dots x_1x_0}{Z_{w/2-1} \quad Z_{w/2-2}}$$

As the MSB x_{w-1} already has a negative weight, the three bits $x_{w-1}x_{w-2}x_{w-3}$ are already Booth encoded. We do not have to pad with two zero bits, nor to perform the $x = 2x - x$ trick on x_{w-1} :

$$X = \underbrace{-x_{w-1}2^{w-1} + x_{w-2}2^{w-2} + x_{w-3}2^{w-3}}_{=Z_{w/2-1}} + \dots$$

This defines $m = \frac{w}{2}$, one less than in the unsigned case.

If w is odd, the most significant Booth digit covers only two MSBs of X :

$$\frac{? \quad x_{w-1}x_{w-2}x_{w-3}x_{w-4}x_{w-5}\dots x_1x_0}{Z_{m-1} \quad Z_{m-2}}$$

The solution here is to sign-extend the number by one bit (marked with '?'). This reduces this case to the previous one (even w). In this case the number m of Booth digits is identical as in the unsigned case.

Then for the partial product generation (Fig. 8.8), Y has to be sign-extended in the case when Y is selected (i.e., for $\text{one}_j = 1$).

Figure 8.10a shows the initial partial product alignment for our 4×4 bit multiplier example, now for the signed case. As $w = 4$ is an even number, we need one partial product less compared to the one in Fig. 8.9. The optimized partial products after sign extension replacement is given in Fig. 8.10b

8.3.3 Higher Radix Multiplication

So far, we discussed radix-2 and radix-4. In the radix-4 case, we showed how to circumvent the unwanted multiplication by 3 using Booth recoding. Further increasing the radix leads to unwanted multiples that cannot be avoided anymore by a simple recoding. In radix-8, with the digit set $\{-4, -3, -2, -1, 0, 1, 2, 3, 4\}$, we again have the unwanted factor 3. For radix-16 the digit set is $\{-8, -7, \dots, 7, 8\}$, containing factors 3, 5, and 7, which all require an addition. A carry-save adder cannot be used, since it

$$\begin{array}{r}
 p_{04} \ p_{04} \ p_{04} \ p_{04} \ p_{03} \ p_{02} \ p_{01} \ p_{00} \\
 + \qquad \qquad \qquad \qquad \qquad \qquad \qquad \text{cpl}_0 \\
 + \ p_{14} \ p_{14} \ p_{13} \ p_{12} \ p_{11} \ p_{10} \\
 + \qquad \qquad \qquad \qquad \qquad \qquad \qquad \text{cpl}_1 \\
 \hline
 = \ p_7 \ p_6 \ p_5 \ p_4 \ p_3 \ p_2 \ p_1 \ p_0
 \end{array}$$

(a) Initial partial products

$$\begin{array}{r}
 \overline{p_{04}} \ p_{03} \ p_{02} \ p_{01} \ p_{00} \\
 + \ \overline{p_{14}} \ p_{13} \ p_{12} \ p_{11} \ p_{10} \qquad \text{cpl}_0 \\
 + \ 1 \qquad 1 \qquad 1 \qquad \qquad \qquad \text{cpl}_1 \\
 \hline
 = \ p_7 \ p_6 \ p_5 \ p_4 \ p_3 \ p_2 \ p_1 \ p_0
 \end{array}$$

(b) Optimized partial products after sign extension replacement

Fig. 8.10 Partial product alignment of a signed 4×4 bit Booth multiplier.

would return each partial product line as two lines (one of sums, one of carries), thus negating the benefit of high-radix recoding. This is the main reason why higher radices are commonly not used. Bewick [Bew94] explored the idea of a radix-8 multipliers where the products $Z_j Y$ are computed in a high-radix carry-save representation (see Sect. 5.2.6), obtained using small ripple-carry adders (RCAs). For radix- 2^k carry-save, the overhead in terms of bit heap size is only one carry bit every k sum bits. The overhead in term of delay is that of a k -bit RCA. This idea could also be explored for radix-16.

8.4 Multiplier Construction for FPGAs

As introduced in Chap. 4.2, many field programmable gate arrays (FPGAs) already provide embedded multipliers in their DSP blocks. Whenever these are available and match the target precision, they provide the most efficient solution for performing multiplications. If not, one of the following situations may require the design of an application-specific multiplier:

1. The size of the required multiplier is much less than the size of the multiplier in the DSP block. One can still map the multiplication to the DSP block by padding the inputs with zeros, but this is a waste of DSP resources. Alternatively, one can use logic-based multipliers [LB18; LBG19], or one can use a single DSP block to compute in parallel several multiplications [Xil17; Lan+19].
2. The size of the required multiplier is larger than the size of the multiplier in the DSP block. Here, several DSP blocks and other logic-based multipliers have to be combined to build a *large multiplier* [DP09; Pas12; Gao+12; Kum+17].

3. The quantity of available DSP blocks is not sufficient (e.g., on a low-cost FPGAs). Here, the missing DSPs have to be replaced with logic-based multipliers.
4. As the position of the DSP blocks on the FPGA die is fixed, the routing delay may be large. At least for small multipliers, it might be faster to use local logic-based multipliers.
5. Not all of the resulting output bits of the multiplier are used. Here, it simply may be more efficient to design a truncated multiplier (see Sect. 8.5) out of DSPs or logic-based multipliers.

The following addresses all these cases.

8.4.1 Using a Single DSP for Multiple Multiplications

Some applications, like the acceleration of the inference in machine learning algorithms, require a small word size, e.g., 8 bits (for more details on machine learning specifics, see Chap. 24). To address this, FPGA manufacturers have described how to perform several small multiplications using the 18×25 DSP blocks of AMD FPGAs [Xil17] or the 18×18 multipliers of Intel FPGAs [Lan+19]. In both cases, one DSP multiplier accommodates two 8×8 multiplications of the form $A \times C$ and $B \times C$, i.e., having one operand in common. We discuss the general idea in the following, starting with the unsigned multiplication case.

8.4.1.1 Unsigned Multiplication

We want to compute $P = A \times C$ and $Q = B \times C$ where A , B , and C are all unsigned values, using one large multiplication

$$R = X \times Y. \quad (8.23)$$

The idea is to pack both operands A and B into a single large bit vector, which is then multiplied by C . To do so, A and B have to be separated by a sufficient amount of zeros, such that their multiplication results do not interfere. Hence, the arguments of the large multiplication are defined as

$$X = A + 2^k B \quad (8.24)$$

$$Y = C, \quad (8.25)$$

with a sufficiently large k .

The result of the multiplication is

$$R = (A + 2^k B) \times C = A \times C + 2^k B \times C. \quad (8.26)$$

The product $P = A \times C$ verifies $0 \leq P < 2^{w_P}$; therefore the two products P and Q do not interfere as soon as $k \geq w_P$. As the word size of A is strictly smaller than 2^{w_P} , the addition of $A + 2^k B$ is just a concatenation of A and B with zeros in between. Figure 8.11 illustrates this for the case when A , B , and C are 8-bit numbers each, using a single 18×27 multiplier (which matches the DSP blocks of recent AMD FPGAs) thanks to $k = w_P$. The two products P and Q can be extracted from the result R of the large multiplication as follows:

$$P = R[w_P - 1 \dots 0] \quad (8.27)$$

$$Q = R[w_P + w_Q \dots w_P]. \quad (8.28)$$

This idea can of course be extended to more than two words, but this will be of limited practical use: multipliers much smaller than 8-bit are probably easier to tabulate in LUTs.

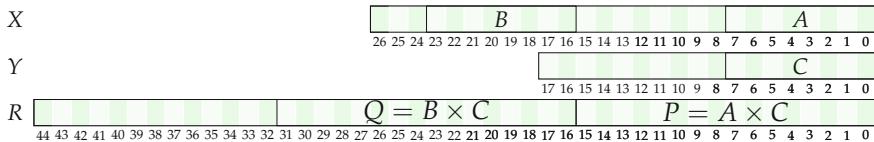


Fig. 8.11 Computing two unsigned 8-bit multiplications $A \times C$ and $B \times C$ using a single 18×27 multiplier.

8.4.1.2 Signed Multiplication

The signed case is a bit more tricky. In the sum $A + 2^k B$, A has to be sign-extended, which requires a real addition. Common DSP blocks allow a pre-addition, i.e., they compute

$$R = (X_0 + X_1) \times Y \quad (8.29)$$

(also see Sect. 4.2, p. 94). Hence, we can set

$$X_1 = A \quad (\text{sign-extended}) \quad (8.30)$$

$$X_2 = 2^k B \quad (8.31)$$

$$Y = C, \quad (8.32)$$

to perform the multiplication of $(A + 2^k B) \times C$.

Again, k has to be chosen to avoid interference. However, this time, the result of the higher product $Q = B \times C$ is affected by the sign bit of $P = A \times$

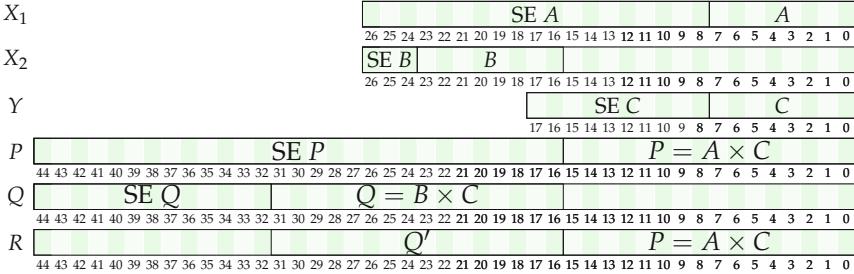


Fig. 8.12 Computing two signed multiplications $A \times C$ and $B \times C$ with 8 bit each using a single 18×27 multiplier with pre-adder.

C. This is illustrated in Fig. 8.12, which also shows the intermediate results P and Q that are not directly accessible. The sign extensions of a variable X are denoted as “SE X ” in Fig. 8.12. While P can be directly recovered by selecting the lower bits of R ,

$$P = R[w_P - 1 \dots 0], \quad (8.33)$$

the other product Q overlaps with the sign extension of P . However, Q can be recovered from

$$Q' = R[w_P + w_Q - 1 \dots w_P]. \quad (8.34)$$

The sign extension of P that overlaps with Q is a bit vector of length w_Q where each bit is equal to the sign bit p_{w_P-1} . Hence, depending on p_{w_P-1} , this vector is either $00 \dots 00 = 0$ or $11 \dots 11 = 2^{w_Q} - 1$, leading to

$$Q' = Q + p_{w_P-1}(2^{w_Q} - 1) \quad (8.35)$$

which provides a means to compute Q out of Q' :

$$Q = Q' - p_{w_P-1}(2^{w_Q} - 1). \quad (8.36)$$

This subtraction can be performed by conditionally adding the opposite of $2^{w_Q} - 1$, which, in two’s complement on w_Q bits, is 1. Thus, the final correction we have to perform is

$$Q = Q' + p_{w_P-1}, \quad (8.37)$$

i.e., we have to add the sign bit of the lower product. The sign bit of P can be simply precomputed as

$$p_{w_P-1} = b_{w_A-1} \oplus b_{w_B-1}. \quad (8.38)$$

It can be input to the post-adder of the DSP block. It can also be pre-added by adjusting the sign extension of the first operand X_1 , using very little additional logic.

8.4.1.3 Extensions

Several results may be accumulated using the post adder if $k > w_P$, in which case k should be selected as large as possible [Xil17]. Lower precisions also allow more multiplications. The implementation of four 4×4 multipliers with shared input, i.e., computing $X_1 \times Y_1$, $X_1 \times Y_2$, $X_2 \times Y_1$, and, $X_2 \times Y_2$, is discussed in the context of deep learning inference accelerators [Xil20].

Conversely [Lan+19], if the DSP block is too small to have $k \geq w_P$ (as is the case if an 18×18 multiplier is to be used for 8×8 multiplication), the technique can still be used with overlapping P and Q . In this case, the least significant bits of Q , those which overlapped with P , have to be computed by a small (LUT-based) multiplier and subtracted from R where they overlap. This removes the overlap, thus restoring the upper bits of P , but also removing any carry that had propagated into the upper bits of Q . These upper bits can then be concatenated to the LUT-computed lower bits.

8.4.2 Multiplier Construction as a Tiling Problem

We now introduce a very useful and generic tool for multiplier design using heterogenous resources, where a multiplier is viewed as an abstract geometrical *tiling* of a rectangle representing the partial product array.

As shown at the beginning of this chapter, a $w_X \times w_Y$ multiplier can be represented as a rectangle of width w_X and height w_Y (see Fig. 8.2). The elementary tiles in this rectangle represent the binary partial products that compute a 1×1 bit multiplication (Fig. 8.13). More generally, we call a *tile* any rectangle of $m \times n$ bits within the larger rectangle of the multiplier. Such

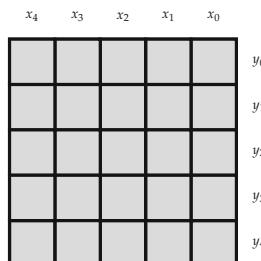


Fig. 8.13 Tiling representation of the binary 5×5 multiplication.

a tile represents the weighted sum of all the partial products it contains, sum which can be computed by a smaller $m \times n$ multiplier.

A multiplier can now be constructed by tiling its rectangle with the shapes of sub-multipliers (tiles), such that every location is covered exactly once. Figure 8.14a shows an example of a valid tiling of a 5×5 bit multiplier that is realized by four rectangular 2×3 and one 1×1 sub-multipliers. The phrase *exactly once* is important here: every location has to be covered, but no overlap is allowed.

Then the large multiplier may be constructed by adding the output bits m_{ij} of all the sub-multipliers M_i corresponding to the tiles, suitably shifted as illustrated by Fig. 8.14b.

To determine these shifts, consider that the LSB of the output of a sub-multiplier is the partial product located at the upper right of a tile. For instance, the LSB of M_2 on Fig. 8.14 is the partial product x_3y_2 , which has weight 2^{3+2} in the partial product sum of the large multiplier. This is therefore also the weight that must be applied to the rest of the product corresponding to M_2 (Fig. 8.14b). Therefore, in the following, we denote the coordinates of a tile with respect to its upper right corner, which also corresponds to the LSB of the corresponding sub-product. The result of a multiplier placed at position (x, y) has the weight 2^{x+y} (it has to be shifted left by $x + y$ bits) in the final sum.

The inputs to the sub-multipliers corresponding to a tile can also be directly read out from the graphical tiling representation: The input ranges of each multiplier input correspond to the projection of the multiplier position to the x and y axis, respectively. For example, the operands of M_2 in Fig. 8.14 are the bit vectors $x_4x_3x_2$ and y_4y_3 .

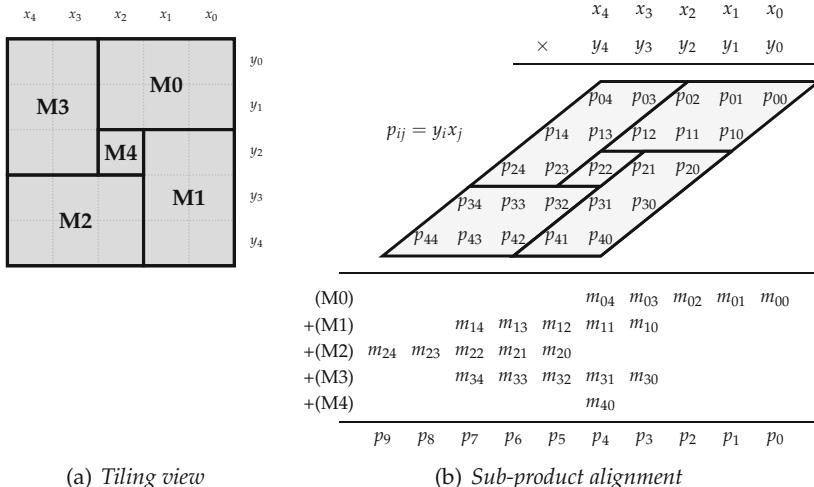


Fig. 8.14 Example of a 5×5 multiplier tiled with 2×3 and 1×1 multipliers.

Remark that tiles can extend outside of the multiplier area, as it corresponds to zero-padding the inputs of the corresponding sub-multipliers. An illustration of this is Fig. 8.19.

For signed multipliers, those sub-multipliers that input the MSB of one of the arguments have to perform a signed multiplication for the corresponding argument. For example, if Fig. 8.14 represents a signed multiplication, then M2 has to be a signed \times signed multiplier, while M1 and M3 have to be signed \times unsigned and unsigned \times signed multipliers.

Note that this tiling approach is compatible with the big picture of Fig. 8.5. In particular, the summation in Fig. 8.14b can be performed by a compressor tree. Tiling is a way to manage the partial product generation step of Fig. 8.5 when heterogeneous resources are available.

Finding the best tiling is an optimization problem which quickly becomes non-trivial if several sub-multipliers of different shapes are allowed. Solving this *tiling problem* in general will be discussed in Sect. 8.4.3. Before that, let us study the various tiles at our disposal in a modern FPGA.

8.4.2.1 Efficiency Metric

We will have to compare various sub-multipliers in order to select the best ones in a given context. For this purpose, we now introduce an *efficiency* metric. As the metric for compressor trees introduced in Sect. 7.3.2.4, it is defined as a benefit-cost ratio:

$$E_s = \frac{\text{area}_s}{\text{cost}_s} \quad (8.39)$$

Here the benefit is the tile area, which measures the amount of multiplication work performed by this tile. The index s is used to distinguish between different tile types (or shapes). Costs considered here for a tile are as follows:

- A DSP cost $\text{cost}_s^{\text{DSP}}$, which is just a count of DSP blocks.
- A LUT cost term $\text{cost}_s^{\text{LUT}}$, which counts the LUTs used to build a sub-multipliers, but also the LUTs required to add its result in the compressor tree. Hence, a DSP also contributes to the LUT cost.

The compressor tree cost depends on other values added in the compressor tree and therefore cannot be defined by an exact formula. Fortunately, as we have seen in Sect. 7.4, the number of LUTs grows fairly linear with the number of bits on the bit heap. We can use this to approximate the cost: Denoting the LUTs to implement the multiplier as $\#LUT_s^{\text{mult}}$, the LUT-cost of adding one bit with a compressor tree as $\#LUT_s^{\text{comp}}$, and the number of output bits as w_s , the total LUT cost of a tile s is defined as

$$\text{cost}_s^{\text{LUT}} = \#\text{LUT}_s^{\text{mult}} + \#\text{LUT}^{\text{comp}} \cdot w_s. \quad (8.40)$$

In the following, we use a value of $\#\text{LUT}^{\text{comp}} = 0.55$ LUTs/bit as obtained for AMD FPGAs in Sect. 7.4.

We also define the efficiencies

$$E_s^{\text{LUT}} = \frac{\text{area}_s}{\text{cost}_s^{\text{LUT}}} \quad \text{and} \quad E_s^{\text{DSP}} = \frac{\text{area}_s}{\text{cost}_s^{\text{DSP}}} \quad . \quad (8.41)$$

8.4.2.2 LUT-Based Sub-multipliers

The simplest, yet efficient, way of constructing LUT-based sub-multipliers is by tabulating the multiplication results in LUTs. Using the common 6-input LUT (LUT6), a 3×3 multiplier can be realized by simply taking one LUT per output bit and tabulating the corresponding results, leading to a resource usage of just six LUT6 for the multiplier, plus the cost to add six bits on the compressor tree. The tile area is $3 \times 3 = 9$ and its efficiency is $E_{3 \times 3} = \frac{9}{6+6 \cdot 0.55} \approx 0.97$.

As discussed in Sect. 4.1, the architectural LUTs of modern FPGAs can be either combined to build larger LUTs or can be broken down into several smaller LUTs sharing some inputs. For illustration, we focus in the following on recent AMD FPGAs, but similar results can be obtained for other FPGAs. The AMD LUT6 can be used as two LUT5 if these two LUT5 share the same inputs (see Sect. 4.1.2). This is the case in a tabulated multiplier. A 2×3 (or 3×2) multiplier can be built using three LUT6 (each computing two LUT5, resulting in one unused LUT5 since the output is 5 bits only). This leads to an efficiency of $E_{2 \times 3} = \frac{6}{3+5 \cdot 0.55} \approx 1.04$, which is better than that of a 3×3 sub-multiplier despite the unused LUT5.

Table 8.3 Properties of multiplier tiles on the AMD target FPGA: LUT-based multipliers (top), multipliers based on LUT+carry chain (middle), DSP-based multipliers (bottom).

Shape	Tile area	$\text{cost}_s^{\text{LUT}}$	$\text{cost}_s^{\text{DSP}}$	E_s^{LUT}	E_s^{DSP}
1×1	1	1.55	0	0.65	–
1×2	2	2.1	0	0.95	–
3×3	9	9.3	0	0.97	–
2×3	6	5.75	0	1.04	–
$2 \times k$	$2k$	$1.55k + 2.1$	0	$\frac{2k}{1.55k+2.1}$	–
24×17	408	22.55	1	18.13	408
Super-tiles (a–d)	816	23.1	2	35.32	408
Super-tiles (e–l)	816	31.9	2	25.58	408

The useful LUT-based sub-multipliers are listed in the top of Table 8.3 and are visualized in Fig. 8.15.

The 1×1 multiplier corresponds to a single AND gate and consumes one LUT. It was added to fill possible gaps during tiling, thus guaranteeing that a solution will always be found. The 1×2 multiplier can be mapped to a single LUT6 (computing two LUT5).

Note that a 1×4 multiplier as used in [KAZ15] can be built from two 1×2 multipliers with the same cost and same efficiency. As a 1×2 multiplier is more generic, the 1×4 multiplier is redundant and will not be used. In [Kak+16], a 4×2 multiplier was used in addition to 3×3 multipliers. As the 4×2 multiplier has a low efficiency of about $E_{4 \times 2} = 0.8$ and its shape can be covered by combining a 3×2 multiplier and a 1×2 multiplier with a higher efficiency, it is not considered here either. Note that all multipliers in Table 8.3 can be also flipped, e.g., a 2×3 multiplier can be used as 3×2 multiplier, leading to additional shapes for the tiling.

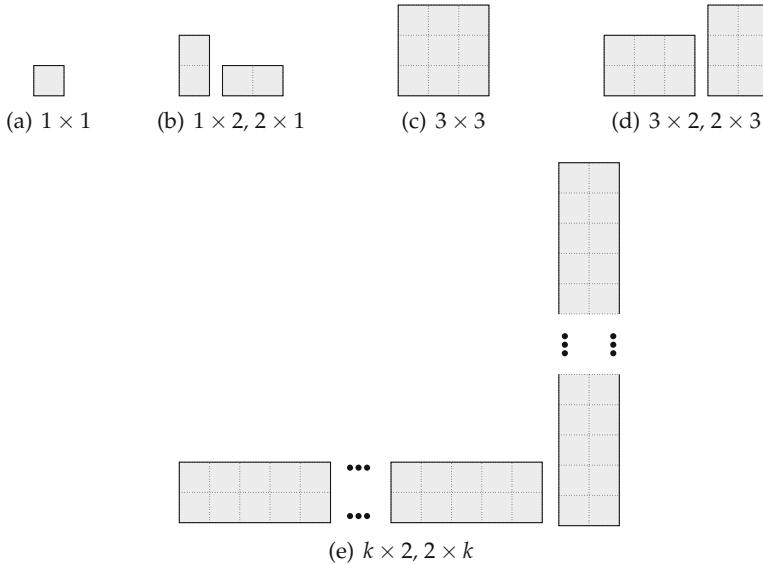


Fig. 8.15 Geometric shapes of the LUT-based tiles (a)–(d) and multipliers based on LUT+carry chain (e).

8.4.2.3 Multiplier Tiles Utilizing the Fast Carry Chain

Besides the LUT-only realizations discussed so far, a multiplier can also exploit the fast carry chain capabilities of modern FPGAs.

In a clever mapping of the classic Baugh-Wooley multiplier array [BW73], two partial bit-products are computed in the LUTs and then directly added by the following carry chain [PI11; KAZ15]. The result is the sum of the two partial product rows, already compressed into a single bit vector. The mapping to a AMD slice is illustrated in Fig. 8.16. This effectively computes a $2 \times k$ or $k \times 2$ multiplication. Its efficiency depends on k as follows: The tiling area is $2k$, while we have $k + 1$ LUTs and $k + 2$ output bits, leading to $E_{2 \times k} = \frac{2k}{(k+1)+(k+2) \cdot 0.55} = \frac{2k}{1.55k+2.1}$. A small k of 6 already leads to an efficiency of $E = 1.05$ which is higher than all the other LUT-based multipliers. For $k \rightarrow \infty$ it reaches up to $E = 1.29$.

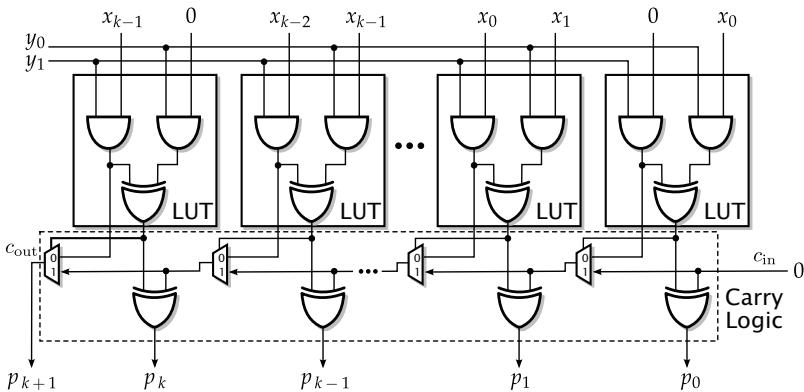


Fig. 8.16 Slice mapping of a $2 \times k$ sub-multiplier [PI11; KAZ15].

A different way to achieve similar results follows the idea of a Booth multiplier, where the Booth recoding as well as the partial product generation are mapped to a single LUT per bit [Wal14; KAZ15; Wal16]. This effectively realizes a $2 \times k$ tile. The advantage is that the fast carry chain is not used for the multiplication, which allows a separate addition. When this addition is used to compress this row and the sum of all the previous rows, one obtains an array multiplier [Wal14; KAZ15; Wal16].

8.4.2.4 DSP-Based Sub-multiplicators

The hardware multipliers included in DSP blocks of modern FPGAs can also be used to build larger multipliers. Their properties have already been discussed in Sect. 4.2. Besides, the DSP blocks in recent FPGAs can be efficiently chained into larger multipliers, thanks to a combination of DSP post-adders and specific routing resources. We call a *supertile* such an efficient chain of DSP blocks [DP09]. For illustration, Fig. 8.17 shows all the possible combinations of two AMD 17 \times 24 DSP48E1 DSP blocks [Xil11] that can

be efficiently chained. Here the constraint is that the DSP products should either be aligned or shifted by exactly 17 bits, the only shift value allowed by the specific routing resource. Due to sharing, this condition translates to the following condition: consider the vector from the upper right corner of one DSP tile to the upper right corner of the other DSP tile. The sum of the coordinates of this vector should be either 0 (no shift) or 17. For example, in Fig. 8.17a this vector has coordinates $(17, -17)$, so the corresponding DSP outputs are aligned (no shift). In Fig. 8.17f, the vector is $(24, -7)$. The sum of its components defines a shift of 17 bits.

For Fig. 8.17b–d, the shift is 0, and their output word size is $w_R = 17 + 24 + 1 = 42$. For Figs. 8.17e–l, the shift is 17 with an output word of size $w_s = 17 + 17 + 24 = 58$.

Of course, this process can be repeated to build larger super-tiles. However, as the DSPs are connected in sequence, the delay or latency (in case of pipelined DSPs) increases with the size of the super-tile. This is why we limit ourselves to small super-tiles consisting of two DSPs.

The properties of the DSP tile as well as the super-tiles are given in the lower part of Table 8.3. The LUT efficiency (E_s^{LUT}) for all DSP-based multipliers is of course very high, as only the compression has to be considered in the LUT cost. Since less bits have to be compressed, the LUT efficiency

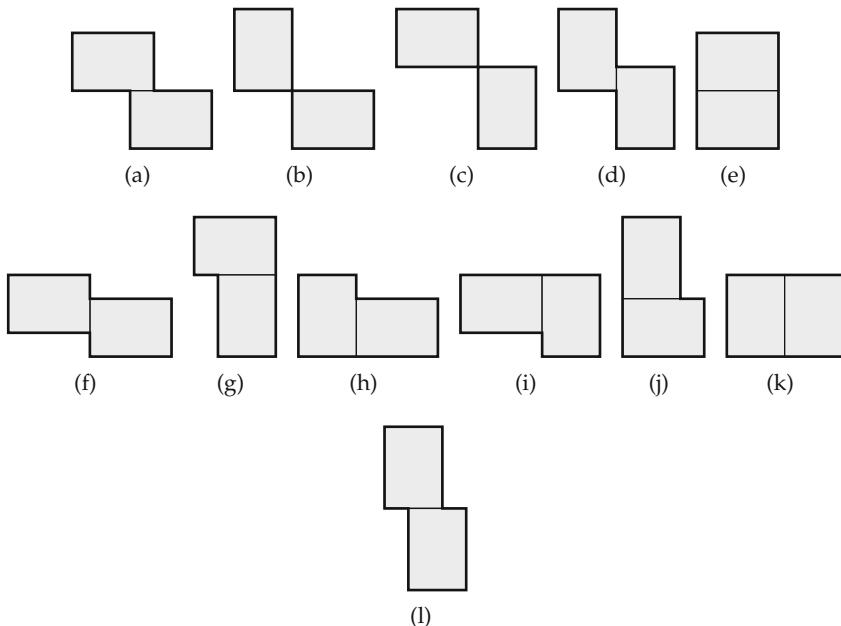


Fig. 8.17 All super-tiles consisting of two 17×24 DSP blocks having a shift of 0 or 17.

is significantly larger for super-tiles. Their DSP efficiency (E_s^{DSP}) is the same for tiles and supertiles, as each DSP contributes with the same area.

8.4.3 Solving the Tiling Problem

This section is dedicated solving the tiling problem. The rules for valid tilings have been introduced in Sect. 8.4.2. We now formalize the corresponding optimization problem.

8.4.3.1 Problem Definition

We assume to have a set of possible sub-multipliers $\mathcal{S} = \{m_0, m_1, \dots, m_{S-1}\}$. Each sub-multiplier m_s is represented as a tile with a different shape. Any of the sub-multipliers discussed above that is suitable to the target technology can belong to this set. Their shape can be arbitrary (super-tiles are examples of non-rectangular tiles, and Karatsuba tiles, introduced in Sect. 8.6, may even consist of non-connected areas). Each sub-multiplier m_s has a cost, denoted as cost_s , that aggregates the corresponding LUT or DSP costs.

Using sub-multipliers from this set, we wish to build a larger multiplier. The shape of this larger multiplier does not have to be rectangular either. In particular, Sect. 8.5 will discuss the construction of so-called truncated multipliers, where the computation of partial products belonging to the upper right corner can be saved. To allow arbitrary shapes within a $w_X \times w_Y$ rectangle, we define a set of binary constants $M_{x,y}$ ($0 \leq x \leq w_X, 0 \leq y \leq w_Y$) such that $M_{x,y}$ is true within the shape of the large multiplier.

The tiling optimization problem can then be formulated as follows:

Tiling Problem: *Given a shape $M_{x,y}$ with $0 \leq x \leq w_X, 0 \leq y \leq w_Y$ of the multiplier and a set \mathcal{S} of sub-multipliers, each associated with cost_s , find a tiling with minimal cost such that each position (x, y) for which $M_{x,y} = 1$ is covered by exactly one sub-multiplier $m_s \in \mathcal{S}$.*

This definition avoids overlaps of sub-multipliers which would lead to incorrect results but allows that a sub-multiplier may overlap with the border of the large multiplier.

It may be difficult to define a sensible cost function combining resources as different as LUTs and DSPs. Hence, a practical variant of the problem would be to constrain the number of DSP blocks and to minimize the remaining logic.

8.4.3.2 Previous Work

Tiling problems have attracted much attention in various fields, from mathematics and computer science (geometry, combinatorics, complexity theory) to industrial processes. A comprehensive survey on the tiling problem until the 1990s is given in [SP92], where the problem is considered as a *packing* and *cutting* problem. The works surveyed span over five decades and various fields, ranging from methods for solving the problem to the analysis of its complexity and some possible applications.

In terms of complexity, variations of the tiling problem have been shown to be *NP-complete* [Lev84] [Lev87]. Tiling a multiplier using square multiplier blocks is a generalization of the problem of [Lev84] and [Lev87]. Hence, the problem can be regarded as NP-complete, too. Tiling can also be seen as a *partitioning problem* which is also NP-complete [Bea+02].

In the context of multiplier design, the first use of tiling was introduced in [DP09] and refined in [Ban+10]. Both works presented heuristics to solve the tiling problem. An optimal solution using ILP was provided in [Kum+17]. This ILP model presented below scales up to multipliers of 64×64 bits. For larger multipliers, improved heuristics that can also treat Karatsuba tiles were proposed in [BKK20].

8.4.3.3 ILP Formulation

To capture that each possible multiplier can be placed at each possible location, binary ILP variables $d_{x,y}^s$ are defined to be true when a sub-multiplier with shape s is located at position (x, y) on the multiplier board. In addition to that, binary constants $m_{x,y}^s$ are defined to be true within the shape s of the sub-multiplier.

With that, the tiling problem can be solved using a single type of constraint. The complete formulation is as follows [Kum+17]:

$$\text{minimize } \sum_{s=0}^{S-1} \sum_{x=0}^{X-1} \sum_{y=0}^{Y-1} \text{cost}_s d_{x,y}^s$$

subject to

$$\text{C1: } \sum_{s=0}^{S-1} \sum_{x'=0}^x \sum_{y'=0}^y m_{x-x', y-y'}^s d_{x', y'}^s = 1 \quad \left. \begin{array}{l} \text{for } 0 \leq x < X, \\ 0 \leq y < Y \\ \text{with } M_{x,y} = 1 \end{array} \right\}$$

The objective is clearly to minimize the total cost, which is just the sum of the costs of the sub-multipliers used.

The constraints in the ILP formulation ensure that the complete shape of $M_{x,y}$ is covered without overlap. To illustrate the constraints, consider

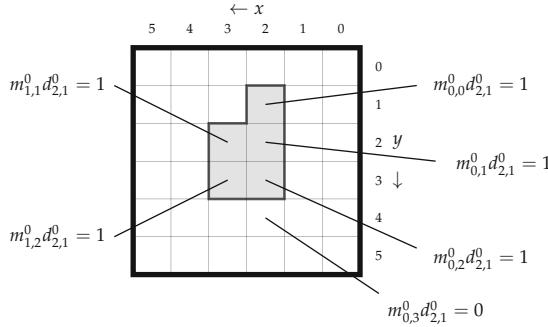


Fig. 8.18 Example placement of a single non-rectangular multiplier tile.

Fig. 8.18, where a single sub-multiplier with shape $s = 0$ is placed on a 6×6 board defined by $M_{0..5,0..5} = 1$. The shape of the sub-multiplier is described by the five non-zero constants $m_{0,0..2}^0 = m_{1,0..1}^0 = 1$; all other $m_{x,y}^0$ are zero. In the example, the sub-multiplier is placed at coordinate $(2, 1)$. As illustrated in Fig. 8.18, there are exactly five coordinates (x, y) , all lying in the area of the sub-multiplier, for which the left-hand side of the corresponding ILP constraint is equal to one. In case no multiplier is located at (x, y) , the result would be zero, while in case that several multipliers overlap at coordinate (x, y) , the result would be >1 . Both of these illegal cases are excluded by the constraints.

Note that due to the binary nature of the variables, the formulation belongs to the class of binary integer linear programming (BILP) problems.

The model can be easily extended by further constraints. For example, it is straightforward to fix the number of total DSPs by the additional constraint

$$\sum_{s=0}^{S-1} \sum_{x=0}^{X-1} \sum_{y=0}^{Y-1} \text{cost}_s^{\text{DSP}} d_{x,y}^s = \# \text{DSP}. \quad (8.42)$$

Remember that $\text{cost}_s^{\text{DSP}}$ just specifies the number of DSP blocks that are contained in a sub-multiplier of shape s (see Table 8.3, p. 229). Similarly, a \leq relation in (8.42) can be used to limit the DSP count to at most $\# \text{DSP}$.

Figure 8.19 shows the exact solution of a tiling using 8 AMD DSPs for a 53×53 multiplier using the multiplier shapes of Table 8.3 (the 53×53 multiplier is the mantissa multiplier in a double-precision floating-point multiplier; see Sect. 11.2). This example illustrates that the optimal tiling may be quite different from a greedy or regular one.

This windmill-like shape allows for sub-multipliers that extend out of the area of the large multiplier. For that, the range of x and y in the ILP formulation has to be extended accordingly (which was not presented so far for simplification). To be precise, the index x (resp. y) should start from a negative value that is the opposite of the largest width (resp. height) of the

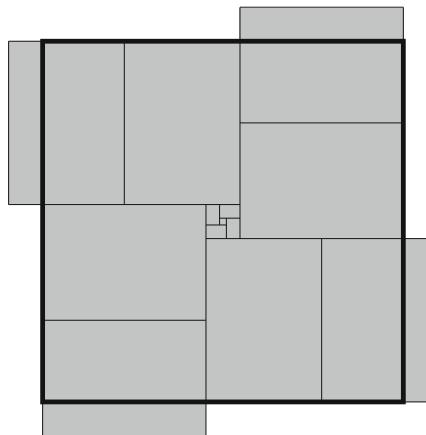


Fig. 8.19 Exact solution of a tiling using 8 DSPs for a 53×53 multiplier.

tiles in \mathcal{S} . The upper range X and Y should be extended by the same value in the positive direction.

Hands on: Integer multipliers

The reader interested in the design of integer multipliers may use the `IntMultiplier` operator of FloPoCo. The command

```
flopoco IntMultiplier wX=24 wY=24
```

produces a 24×24 multiplier using the default tiling method, which is a heuristic based on beam search [BKK20] but can be changed with the global option `tiling`. Its value may be optimal for the ILP-based solver [Kum+17] described above. A fast greedy search [BKK20] can be selected with `heuristicGreedyTiling`.

Several further options exist for `IntMultiplier`. The `dspthreshold` parameter controls the ratio a DSP block may overlap. For example, `dspthreshold=0` only allows DSPs that are fully used, while `dspthreshold=1` would use a DSP block even to cover a 1×1 area. Apart from that, the DSP count may be limited by the `maxDSP` parameter. This allows fine control over the LUT/DSP tradeoff.

For example, the command

```
flopoco IntMultiplier wX=24 wY=24 tiling=optimal maxDSP=1
```

produces a solution using 1 DSP as shown in Fig. 8.20.

The following command produces the tiling shown in Fig. 8.19:

```
flopoco IntMultiplier wX=53 wY=53 tiling=optimal maxDSP=8
```

Figures such as Figs. 8.19 and 8.20 can be obtained with `generateFigures=true`.



Fig. 8.20 Exact solution of a tiling using 1 DSPs for a 24×24 multiplier.

8.5 Truncated Multipliers

As introduced in Sect. 8.1, there are numerous cases where not all the output bits from the final product are required. A truncated multiplier is defined by $\ell_P > \ell_X + \ell_Y$. The most common case of truncated multiplication is a square multiplication where the output has the same format as both inputs, or $w_X = w_Y = w_P = w$. To avoid the issues of saturation (which can be handled separately), we illustrate this section with products of fixed-point numbers in $[0, 1)$, in which case the format of X , Y , and P is $\text{ufix}(-1, \ell_P)$ with $\ell_P = -w$. We keep the notation ℓ_P so that all the following can still be used if $\ell_P \neq (\ell_X, \ell_Y)$.

A first architecture consists of an exact multiplier followed by a truncation or a rounding. Actually, the rounding step can be merged inside the exact multiplier: rounding is performed by adding a constant 1 to the bit array at position $\ell_P - 1$. Then, truncating the result of the compression at position ℓ_P provides a product correctly rounded to the nearest, with ties up. A fixed-point correctly rounded multiplier therefore has almost the same area and delay as an exact one, the difference being one constant bit in the bit array.

8.5.1 Plain or Booth Truncated Multipliers

The previous architecture is usually not called a truncated multiplier. In a truncated multiplier, the accuracy specification is relaxed (in this book, to a faithfully rounded result as defined in Sect. 3.1.8). Then many of the partial products do not have to be computed at all, thus improving resources, delay, and energy.

There has been much research on truncated multipliers [SS93; KS97; MTV06; Pet+10; Ban+10; KH11; DRC14], but for the purpose of this book, a truncated multiplier is really an instance of the general case of truncated bit heap studied in Sect. 7.2.4—indeed, the basic truncated multiplier was taken as an example there (see Fig. 7.17). The reader is therefore referred to this section for the technical details. The generic bit-heap truncation algorithm given there as Listing 7.1 applies to the basic radix-2 multiplier, but also to Booth-reduced bit heaps. Figure 8.21 compares the bit heaps obtained in both cases.

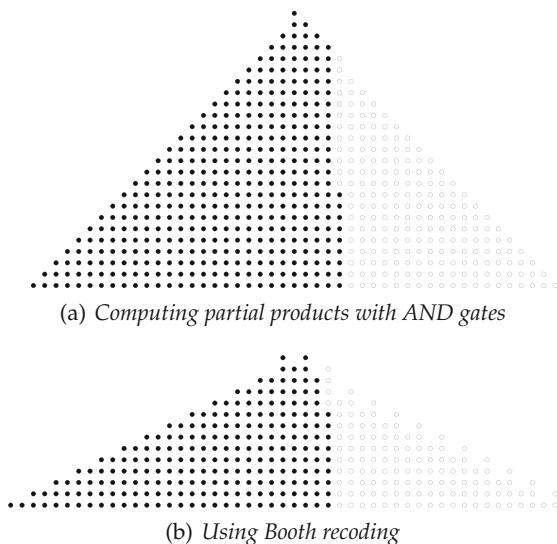


Fig. 8.21 24-bit faithful truncated multipliers (constant bits not shown).

Table 8.4 gives the number $g = \ell - \ell_{\text{ext}}$ of extra columns that must be kept in a faithful truncated multiplier in both cases. Note that this table is equally valid for signed and unsigned multipliers, since the architectural difference between signed and unsigned only affects the leading bits of the result, which are unconcerned by truncation.

A formula that summarizes the faithful truncation of plain multipliers seems to be [A005126] that g guard bits are enough for $w < 2^g + g$. We also observe in Table 8.4 that Booth recoding typically allows to truncate one column more. This is easily explained by the observation that Booth recoding divides by two Δ^{minor} , the weighted sum of the bits in the truncated part.

We finish this section with a description of an extension of the tiling-based ILP approach that manages both the last-column reduction and the constant bits.

Table 8.4 Numbers of guard bits $g = \ell - \ell_{\text{ext}}$ for a faithful truncated unsigned multiplier for $\text{ufix}(-1, \ell_P)$.

Plain multiplier		Booth multiplier	
$-\ell_P = w$	g	$-\ell_P = w$	g
3 to 5	2	3 to 6	2
6 to 10	3	7 to 15	3
11 to 19	4	16 to 32	4
20 to 36	5	33 to 65	5
37 to 69	6	66 to 130	6
70 to 134	7	131 to 259	7
135 to 263	8	260 to 516	8
264 to 520	9	517 to 1029	9

8.5.2 Tiling for Optimal Truncated Multipliers on FPGAs

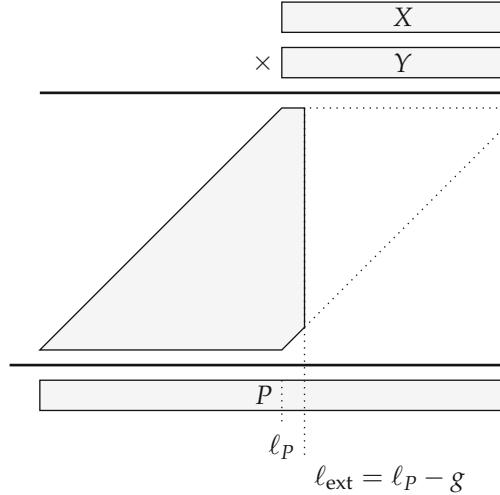
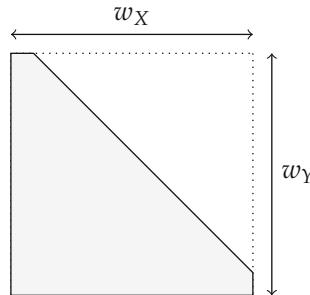
The truncation of the bit heap as described above works well for the basic and Booth-recoded multipliers, as their generation of the partial products is fixed. In case we have several options to produce the partial products like in the tiling method for FPGAs (see Sect. 8.4.2), there are many possible solutions that also lead to different bit heaps. This degree of freedom can be used to further optimize the cost by searching for the best tiling for the truncated multiplier.

In the simplest form, the truncated multiplier can be represented by modifying the multiplier shape that has to be tiled. Figure 8.22 illustrates how the truncation line at $\ell_{\text{ext}} = \ell_P - g$ in the partial product alignment of Fig. 8.22a translates to the modified tiling area shown in Fig. 8.22b. Indeed, thanks to the constants $M_{x,y}$ introduced in Sect. 8.4.3, we can define arbitrary multiplier shapes. Figure 8.23 shows an example tiling solution of a 32×32 multiplier truncated to 32 output bits using one DSP block and the ILP method introduced in Sect. 8.4.3.3 with the tiles of Fig. 8.15, p. 230.

Such a tiling is guaranteed to meet the truncation error constraint (7.14) if ℓ_{ext} was determined correctly. However, it might not give an optimal result:

- It does not attempt to truncate bits from the rightmost column.
- As highlighted in [Ban+10], a sub-multiplier covering parts of non-required multiplier area (the dotted area in Fig. 8.22b) might compensate some error when removing partial products of the required multiplier area (gray area in Fig. 8.22b).

A solution to both problems is to let the ILP problem directly select which parts of the tiling area should be left uncovered to ensure the error constraint [BKD21]. Still, we start with the truncated shape of Fig. 8.22b because it reduces the problem size.

(a) *Partial product alignment*(b) *Resulting tiling area***Fig. 8.22** A truncated multiplier using guard bits.

The following shows how to integrate these ideas in the tiling ILP model of Sect. 8.4.3.3 to globally solve the truncated tiling optimization problem.

To get there, the idea is to modify the ILP formulation of Sect. 8.4.3.3 to allow for uncovered area, but take into account the corresponding error. To do so, we first allow uncovered areas by changing the constraints to

$$\text{C1: } \sum_{s=0}^{S-1} \sum_{x'=0}^x \sum_{y'=0}^y m_{x-x', y-y'}^s d_{x', y'}^s = b_{x,y} \quad \left. \begin{array}{l} \text{for } 0 \leq x \leq X, \\ 0 \leq y \leq Y \\ \text{with } M_{x,y} = 1 \end{array} \right\} .$$

Instead of requiring that each position is covered by exactly one tile, we allow it to be not covered by introducing new variables $b_{x,y}$. So, $b_{x,y}$ is in fact equal to the number of tile overlappings at position (x, y) . When $b_{x,y} = 0$,

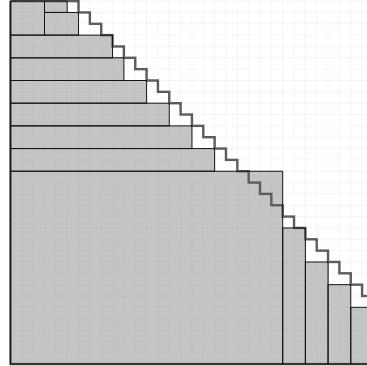


Fig. 8.23 Optimal solution of a tiling area satisfying a 32×32 multiplier truncated to 32 output bits.

this corresponds to an uncovered position and a result \tilde{P} that is less or equal to the correct result P and, hence, contributes a negative error. When $b_{x,y} = 1$ this corresponds to no error. One could even allow multiple overlapping tiles by allowing $b_{x,y} > 1$. As this is very unlikely to appear in the optimal solution and would complicate the model, we do not allow overlappings by constraining $b_{x,y} \in \{0, 1\}$.

Now, the maximum absolute error introduced at that coordinate is

$$\bar{\delta}_{x,y} = (1 - b_{x,y}) \cdot 2^{x+y}, \quad (8.43)$$

leading to a total maximum absolute error of

$$\bar{\delta}_{\text{tiling}} = \sum_{x=0}^{X-1} \sum_{y=0}^{Y-1} \bar{\delta}_{x,y}. \quad (8.44)$$

As this has a negative contribution to the total error, we can make use of the constant correction trick [SS93] as introduced in Sect. 7.2.4.3. Hence, we add a constant C to the bit heap which lifts the error to both positive and negative. The total error now gets

$$|C - \delta_{\text{tiling}}| \leq \bar{\delta}_{\tilde{P}}. \quad (8.45)$$

with δ_{tiling} being bound by $0 \leq \delta_{\text{tiling}} \leq \bar{\delta}_{\text{tiling}}$. To obtain a faithfully rounded result, it has to be less than $\bar{\delta}_{\tilde{P}} = 2^{\ell_P - 1}$ (see Sect. 7.2.4.1).

To make (8.45) linear, to be solvable by ILP, we have to consider two cases:

1. $C \geq \delta_{\text{tiling}}$: This leads to the positive error $C - \delta_{\text{tiling}}$ which is maximal for $\delta_{\text{tiling}} = 0$, leading to the constraint

$$C < \bar{\delta}_{\tilde{P}} \quad (8.46)$$

2. $C < \delta_{\text{tiling}}$: This leads to a negative error which has to be negated by $-(C - \delta_{\text{tiling}})$. This term is maximal for $\delta_{\text{tiling}} = \bar{\delta}_{\text{tiling}}$, leading to the constraint

$$\bar{\delta}_{\text{tiling}} - C < \bar{\delta}_{\tilde{P}} \quad (8.47)$$

Note that each case does not constrain the other case as (8.46) only constrains the maximum of C , while (8.47) constraints the maximum of $\bar{\delta}_{\text{tiling}}$.

The final constraints for our ILP are now obtained by substituting (8.43) and (8.44) into (8.46) and (8.47) to obtain

$$\text{C2:} \quad C < \bar{\delta}_{\tilde{P}} \quad (8.48)$$

$$\text{C3:} \quad \sum_{x=0}^{X-1} \sum_{y=0}^{Y-1} (1 - b_{x,y}) \cdot 2^{x+y} - C < \bar{\delta}_{\tilde{P}} \quad (8.49)$$

Note that C1 can be substituted into C3 which is not done here for brevity.

The last missing piece is to consider the constant C in the cost function. For that, we have to count the non-zero bits of C , which can be done by introducing the individual bits of C as binary variables $c_i \in \{0, 1\}$ in the model, which are constrained to be

$$\text{C4:} \quad C = \sum_{i=l_C}^{m_C} 2^i c_i \quad (8.50)$$

where its MSB and LSB have to be chosen to $m_C = \ell - 2$ and $l_C = \ell_{\text{ext}}$ (see discussion in Sect. 7.2.4.3), respectively.

The overall objective can be simply extended by adding the compressor tree cost $\#\text{LUT}^{\text{comp}}$ for each non-zero bit c_i . The overall ILP model becomes

$$\begin{aligned} \text{minimize} \quad & \sum_{s=0}^{S-1} \sum_{x=0}^{X-1} \sum_{y=0}^{Y-1} \text{cost}_s d_{x,y}^s + \sum_{i=l_C}^{m_C} \#\text{LUT}^{\text{comp}} c_i \\ \text{subject to} \quad & \text{C1, C2 and C3.} \end{aligned}$$

Figure 8.24 shows a tiling solution for the example already used in Fig. 8.23, but now using the refined model introduced in this section. The outer shape of Fig. 8.23 is shown as a thick line to illustrate that a similar shape was found by only constraining the maximum error. However, some coordinates inside the area surrounded by the thick line are not covered, while other parts inside this area are covered to compensate for this error. By doing so, the $2 \times k$ tiles can be used much more efficient as many of the less efficient 1×1 tiles used in Fig. 8.23 are avoided.

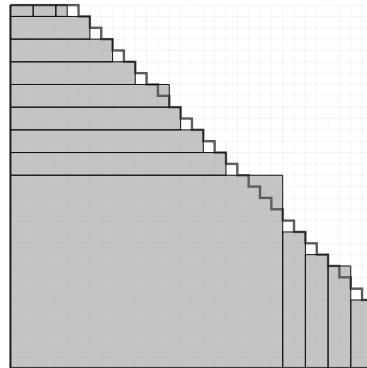


Fig. 8.24 Globally optimal truncated multiplier tiling solution of a 32×32 multiplier truncated to 32 output bits.

The solution in Fig. 8.24 requires a compensation constant of $C = 2013265920$, which requires to add four constant ones to the bit heap. The tiling error can be obtained by summing up the error 2^{x+y} for each (x,y) that is not covered, leading to $\bar{\delta}_{\text{tiling}} = 4,152,360,961$. The total error is

$$|C - \delta_{\text{tiling}}| = 1,954,545,664, \quad (8.51)$$

which is just less than the maximum allowed error of

$$\bar{\delta}_{\tilde{P}} = 2^{\ell_P-1} = 2^31 = 2,147,483,648. \quad (8.52)$$

Shortening any of the $2 \times k$ tiles in Fig. 8.24 would increase the error by at least $2^{27} + 2^{28} = 402,653,184$ and would exceed the error bound.

Hands on: Truncated integer multipliers

Truncated multipliers are obtained with FloPoCo whenever a lower output word size is specified than required by using the optional argument `wOut`:

```
flopoco IntMultiplier wX=32 wY=32 wOut=32
```

produces a 32×32 multiplier with the output truncated to 32 bits as well.

It uses a pre-defined tiling area using the guard bit approach. This works with any tiling method.

The optimal truncated tiling method of Sect. 8.5.2 can be selected by passing `tiling=optimal`.

8.6 The Karatsuba Method

Karatsuba's method [KO63] was the first method for computing products with sub-quadratic complexity. It trades (sub-)multiplications for additions, which are less expensive.

8.6.1 Two-Part Splitting

Consider a large multiplication of size $2w \times 2w$ bits. It can be split into four smaller multiplications of size $w \times w$ thanks to the tiling depicted in Fig. 8.26a.

Mathematically, each input is split into two smaller numbers, each of size w , as illustrated in Fig. 8.25. The product can then be rewritten

$$X \times Y = (X_1 2^w + X_0)(Y_1 2^w + Y_0) \quad (8.53)$$

$$= \underbrace{X_1 Y_1}_{=M4} 2^{2w} + (\underbrace{X_1 Y_0}_{=M3} + \underbrace{X_0 Y_1}_{=M2}) 2^w + \underbrace{X_0 Y_0}_{=M1}. \quad (8.54)$$

The basic idea of Karatsuba's algorithm is based on the identity:

$$X_1 Y_0 + X_0 Y_1 = (X_0 + X_1)(Y_0 + Y_1) - \underbrace{X_0 Y_0}_{=M1} - \underbrace{X_1 Y_1}_{=M4} \quad (8.55)$$

Hence, by reusing the sub-products of M1 and M4, we can compute the sum of the results of M2 and M3 by a single multiplication: this reduces the number of sub-multiplications from four to three, saving one sub-multiplier. The price to pay are two new additions computing $X_0 + X_1$ and $Y_0 + Y_1$ (called pre-additions in the following) and two additional negative terms in the final sum (called post-additions). Besides, the new product is a $(w+1) \times (w+1)$ -bit one, hence requiring a slightly larger multiplier than each of the $w \times w$ -bit products it replaces.

$$\begin{aligned} X &= \begin{array}{c} \xleftarrow[w \text{ bits}]{\times} \xrightarrow[w \text{ bits}]{\times} \\ \boxed{\begin{array}{|c|c|} \hline X_1 & X_0 \\ \hline \end{array}} \end{array} = 2^w X_1 + X_0 \\ Y &= \boxed{\begin{array}{|c|c|} \hline Y_1 & Y_0 \\ \hline \end{array}} = 2^w Y_1 + Y_0 \end{aligned}$$

Fig. 8.25 Two-part splitting of both X and Y .

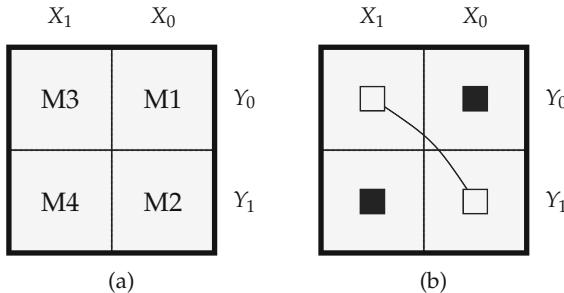


Fig. 8.26 Graphical tiling representation of two-part Karatsuba.

Area-wise, Karatsuba's method reduces the cost for large enough values of w , exactly how large is technology-dependent. Delay-wise, post-additions may be performed in constant time using carry-save arithmetic. Equivalently, they can be merged with small overhead in the final bit array compression. Pre-additions, on the other hand, have a significant timing overhead due to the need for carry propagation. Therefore, Karatsuba's method will typically improve area, but degrade timing.

8.6.2 Subtractive Karatsuba Formula

An alternative form of (8.55) is

$$X_1Y_0 + X_0Y_1 = (X_0 - X_1)(Y_1 - Y_0) + M1 + M4. \quad (8.56)$$

It uses pre-subtractions, which have similar hardware cost as pre-additions (see Sect. 5.2.4). However, the product $(X_0 - X_1)(Y_1 - Y_0)$ is now a product of signed numbers (even to implement an unsigned large multiplication). In other words, the pre-subtractions add one bit, just like pre-additions, but this extra bit added by is now a sign bit.

On some FPGAs, the DSP multipliers are designed for signed multiplication and can only be used for unsigned products by forcing the sign bit to 0, thus losing one bit. In this case, (8.56) will be the preferred formulation: it will allow us to exploit the sign bit inputs of the DSP blocks that would otherwise be wasted.

All the following can use either the additive (8.55) or subtractive (8.56) formula, but we present it with the additive version for simplicity.

8.6.3 Tile Representation

We will now give an interpretation of Karatsuba's method as a special tile of the tiling representation introduced in Sect. 8.4.2. Figure 8.26a shows the tiling representation of the 2-part splitting described in (8.53). Each of the four sub-products appears as a tile. When two multipliers are replaced by a single one using Karatsuba's method, they are represented using linked empty squares as shown for M2 and M3 in Fig. 8.26b. The two other sub-products that are reused for this (M1 and M4) are represented using black squares [Pas12; Kum+18].

This representation defines a new kind of tile linking three sub-multipliers to construct larger multipliers using the tiling approach. It will prove useful when generalizing Karatsuba's formula.

8.6.4 Square K-Part Splitting

The inputs to a large multiplication may also be split into K segments of w bits [Mon05]:

$$X \times Y = \left(\sum_{i=0}^{K-1} 2^{iw} X_i \right) \times \left(\sum_{j=0}^{K-1} 2^{jw} Y_j \right) \quad (8.57)$$

$$= \sum_{i,j} 2^{(i+j)w} X_i Y_j \quad (8.58)$$

Now, in (8.58), any expression of the form $X_i Y_j + X_j Y_i$ may be replaced by

$$X_i Y_j + X_j Y_i = (X_i + X_j)(Y_j + Y_i) - X_j Y_j - X_i Y_i, \quad (8.59)$$

which is the generalized form of (8.55). Again, (8.59) computes the sum of two tile sub-products using only one multiplier. The novelty is that diagonal sub-products $X_i Y_i$ may be reused more than one time, which further improves the efficiency of the method.

Take, for example, a $3w \times 3w$ multiplier where the arguments are split into three parts, i.e., we perform a 3-part splitting

$$X \times Y = (X_2 2^{2w} + X_1 2^w + X_0)(Y_2 2^{2w} + Y_1 2^w + Y_0) \quad (8.60)$$

$$\begin{aligned} &= X_0 Y_0 \\ &\quad + (X_0 Y_1 + X_1 Y_0) 2^w \\ &\quad + (X_0 Y_2 + X_2 Y_0 + X_1 Y_1) 2^{2w} \\ &\quad + (X_1 Y_2 + X_2 Y_1) 2^{3w} \\ &\quad + X_2 Y_2 2^{4w} \end{aligned} \quad (8.61)$$

$$\begin{aligned} &= X_0 Y_0 + X_1 Y_1 2^{2w} + X_2 Y_2 2^{4w} \\ &\quad + ((X_0 + X_1)(Y_0 + Y_1) - X_0 Y_0 - X_1 Y_1) 2^w \\ &\quad + ((X_0 + X_2)(Y_0 + Y_2) - X_0 Y_0 - X_2 Y_2) 2^{2w} \\ &\quad + ((X_1 + X_2)(Y_1 + Y_2) - X_1 Y_1 - X_2 Y_2) 2^{3w}. \end{aligned} \quad (8.62)$$

Here, (8.59) could be used three times, which reduces the sub-multiplier count from $3 \times 3 = 9$ to only 6.

It is possible to describe (8.59) as a geometric tiling rule: Whenever two sub-multipliers of identical size appear at position (i, j) and (j, i) they can be replaced by a single multiplier when multipliers of the same size are available at (i, i) and (j, j) .

For the 3-part splitting of (8.62), this is illustrated in Fig. 8.27a. The black squares again show the multipliers whose results are reused by the linked empty squares. To count the number of sub-multipliers, one has to count the number of linked pairs, added to the number of black squares. The number of linked pairs also directly gives the number of saved sub-multipliers compared to the straightforward solution. While for $K = 3$ it is relatively easy to derive (8.62) manually, it is more complicated for larger K , whereas it remains easy to solve graphically, as illustrated in Fig. 8.27b and c for 4-part and 6-part splitting, respectively.

In general, K -Part splitting saves a triangle of $K(K - 1)/2$ multiplications. For instance, in the 4-part splitting of a $4w \times 4w$ multiplier shown in Fig. 8.27b, 6 of the 16 required $w \times w$ bit multiplications can be saved.

8.6.5 Recursive Karatsuba

The multiplication schemes above can also be applied recursively (it was actually the original formulation [KO63]), i.e., each multiplier is split into smaller multiplications where each of the smaller multiplication is itself split into smaller multiplications, etc. This is traditionally used in software for fast multiple-precision arithmetic [GMPweb; BZ10]: (8.55) is applied recursively until the smaller multipliers match the capabilities of the hardware multipliers offered by the processor.

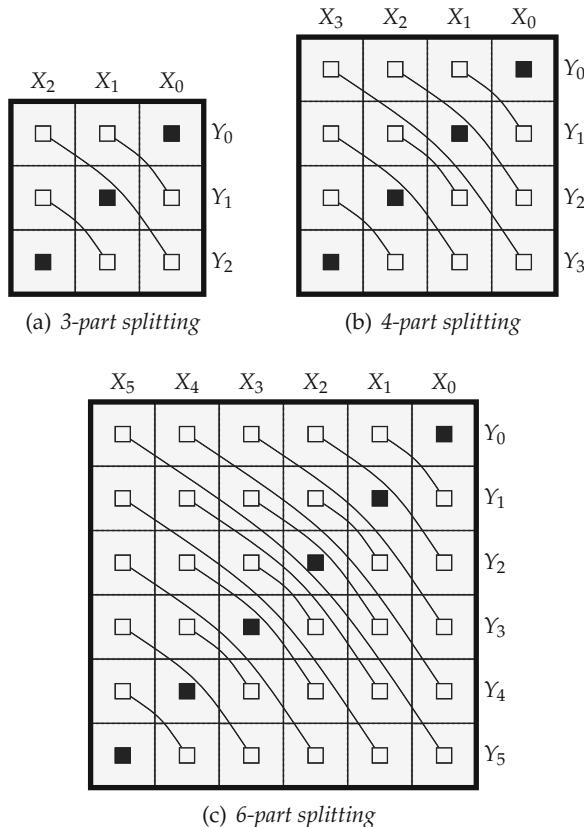


Fig. 8.27 Tilings using the Karatsuba K -part splitting with square multipliers.

Consider the $4w \times 4w$ multiplication again. It can be realized by using three multiplications of up to $(2w+1) \times (2w+1)$ using the 2-part splitting in (8.55). Each of these $(2w+1) \times (2w+1)$ multiplications can then be realized by using three $(w+1) \times (w+1)$ multiplications by applying another 2-part splitting. This leads to a total of only nine $(w+1) \times (w+1)$ multiplications (compared to ten in the 4-part splitting). However, this comes at the price of additional critical path delay or latency, as two pre-additions have to be performed in sequence.

In other words, in hardware, the choice between K-part splitting or recursive Karatsuba exposes a trade-off between latency and number of multipliers.

8.6.6 Generalized Karatsuba Formula

One observation from the tiling representation is that the pattern that results from Karatsuba's method can be arbitrary shifted on the rectangle (like any other sub-multiplier). In other words it is possible to pair any two tiles that share the same weight: Taking two multiplications $X_i Y_j$ and $X_k Y_\ell$, as soon as $i + j = k + \ell = s$, the complete product involves $2^{sw}(X_i Y_j + X_k Y_\ell)$, and it is therefore possible to use the rewriting

$$X_i Y_j + X_k Y_\ell = (X_i + X_k)(Y_j + Y_\ell) - X_i Y_\ell - X_k Y_j \quad (8.63)$$

or

$$X_i Y_j + X_k Y_\ell = (X_i - X_k)(Y_j - Y_\ell) + X_i Y_\ell + X_k Y_j \quad (8.64)$$

assuming $X_i Y_\ell$ and $X_k Y_j$ are already computed [Kum+18]. Note that these formulae generalize (8.59) and its subtractive version.

Graphically, this can be viewed as applying Karatsuba's method on a sub-square of the large multiplier. For square multipliers, this generalization is in principle less interesting than the classic K -part decomposition, because the latter exposes more reuse of the diagonal subproducts. However, it could be used to apply Karatsuba's technique to non-square large multipliers: rectangular ones or truncated multipliers, for instance. It is also interesting for FPGAs providing rectangular multipliers, as we discuss next.

8.6.7 Karatsuba with Rectangular Sub-multiplicators

Karatsuba's method so far reduces a large multiplication to small square multipliers. For FPGAs whose DSP blocks provide rectangular multipliers (see Sect. 4.2), one option is to use them as square ones (e.g., using a 17×24 -bit DSP block as a 17×17 -bit multiplier), but this is obviously a waste of DSP resources.

This section presents an alternative way to use DSP blocks more efficiently [Kum+18; BKK20]. It is illustrated with the example of the DSP48E1 DSP blocks found in AMD FPGAs. For simplicity, we consider these as 17×24 -bit unsigned multipliers, but the following derivation can be easily adapted for signed multipliers.

The starting point is the generalized Karatsuba formula (8.63), as presented in Sect. 8.6.6. For the 17×24 DSP, the naive way would be to split X into $w_X = 17$ bit chunks and Y into $w_Y = 24$ bit chunks. To apply (8.63), the two products $X_i Y_j$ and $X_k Y_\ell$ must have the same weight. The weight of $X_i Y_j$ is $2^{17i+24j}$, and the weight of $X_k Y_\ell$ is $2^{17k+24\ell}$; therefore the constraint is $17i + 24j = 17k + 24\ell$ or

$$17(i - k) = 24(\ell - j). \quad (8.65)$$

As 17 and 24 do not have any divisor in common, $i - k$ must be a multiple of 24, and $\ell - j$ must be a multiple of 17 to benefit from Karatsuba's method. So, the smallest multiplier that can benefit from this has an argument size of $(24 + 1) \times (17 + 1) = 450$ bits, i.e., a 450×450 multiplier on which one (!) DSP could be saved out of $24 \times 17 = 408$ DSPs in total. Of course, this is not very satisfying. The trick is therefore to find splittings of the two inputs that will have a much larger common divisor [Kum+18]. This will be detailed next.

Consider the general case of splitting argument X in w_X -bit chunks, and argument Y into w_Y -bit ones. The weight of $X_i Y_j$ is now $2^{w_X i + w_Y j}$. Therefore, for $X_i Y_j$ and $X_k Y_\ell$ to share the same weight, we must have $w_X i + w_Y j = w_X k + w_Y \ell$ for $i \neq k$ and $j \neq \ell$. We can rewrite this as

$$w_X N - w_Y M = 0 \text{ with } N, M \in \mathbb{N}. \quad (8.66)$$

Now, the smallest values of N and M to satisfy (8.66) are obtained by $M = w_X/w$ and $N = w_Y/w$ where w is the greatest common divisor (GCD) of w_X and w_Y , i.e.,

$$w = \gcd(w_X, w_Y).$$

Hence, the first common weight that appears is 2^{wNM} . Thus, the smallest multiplier for which Karatsuba's method can be applied is of size $wM(N + 1) \times wN(M + 1)$. As a large w allows for both small N and M , the product wNM becomes smallest for the largest possible w .

Taking the 17×24 -bit multiplier case again, the GCD is one, leading to the unfavorable case of $wNM = w_X w_Y$. The GCD gets maximal for the standard Karatsuba case of $w_X = w_Y = w$. However, this means using the 17×24 -bit multiplier as a 17×17 -bit multiplier, heavily underutilizing the DSP block.

If we accept to use the slightly smaller tile size of 16×24 (which means under-utilizing the DSP blocks, but much less than if we use them as 17×17), then we get a much larger GCD of $w = \gcd(16, 24) = 8$, with $M = 16/8 = 2$ and $N = 24/8 = 3$. The smallest multiplier for which we may apply Karatsuba's method is now $8 \cdot 2 \cdot (3 + 1) \times 8 \cdot 3 \cdot (2 + 1) = 64 \times 72$. This specific example is shown in Fig. 8.28a. On this figure, we made the choice that the index of each X chunk is $2i$, and the index of each Y chunk is $3j$, which makes it easier to spot the situations where $2i + 3j = 2k + 3\ell$.

A natural question to answer is: are there other possibilities of rectangular tilings like the 16×24 tiling discussed above that would lend themselves to Karatsuba pairing? We considered the following tiles:

- 18×24 (with one line of the multiplication performed using logic resources). This leads to $w = \gcd(18, 24) = 6$, $M = 3$, and $N = 4$.
- 15×25 (under-using the DSP in one dimension and again with one line of the multiplication performed using logic resources). This leads to $w = 5$, $M = 3$, and $N = 5$.

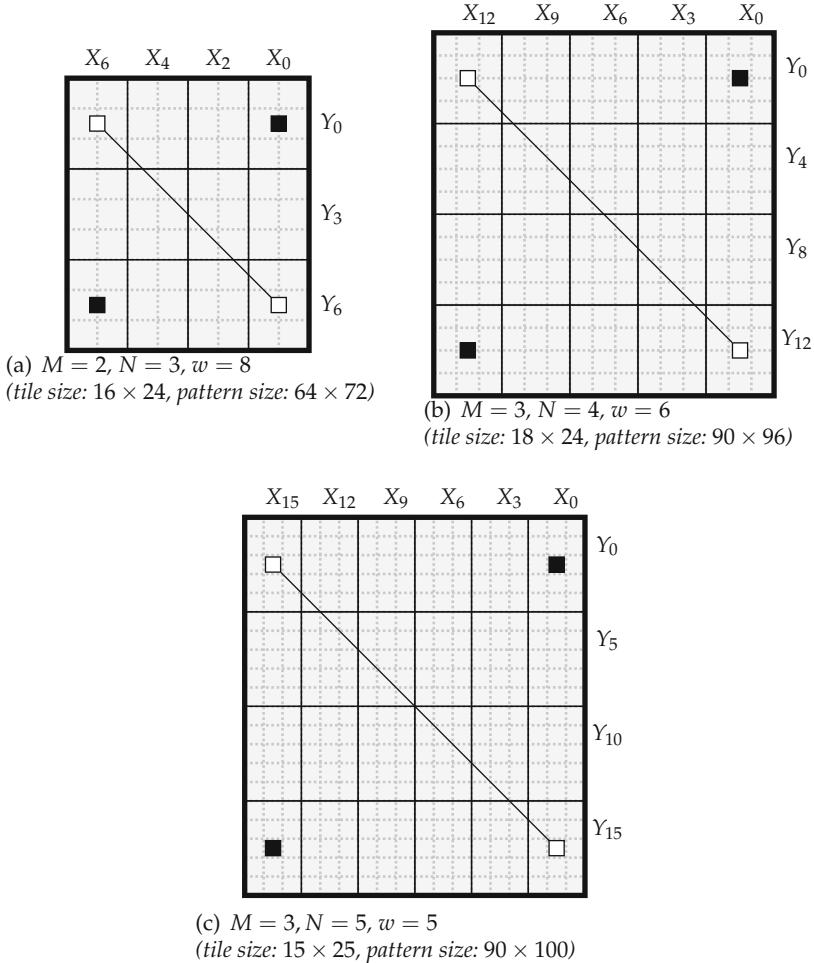


Fig. 8.28 Minimal Karatsuba patterns with various rectangular tiles matching AMD DSP blocks.

Figure 8.28b–c show the smallest possible multipliers on which Karatsuba’s method can be applied with these rectangular tiles. We conclude that the opportunities for Karatsuba’s method are higher for the 16×24 configuration and select this tile for actual evaluation in the rest of this section.

Figure 8.29 shows the tile representation of the first three K -part splittings for the 16×24 configuration. As before, two multipliers with linked empty squares are replaced by a single one. In contrast to the square cases (Fig. 8.27), we have “gaps” between the different multipliers. These gaps can be filled with other accordingly shifted patterns.

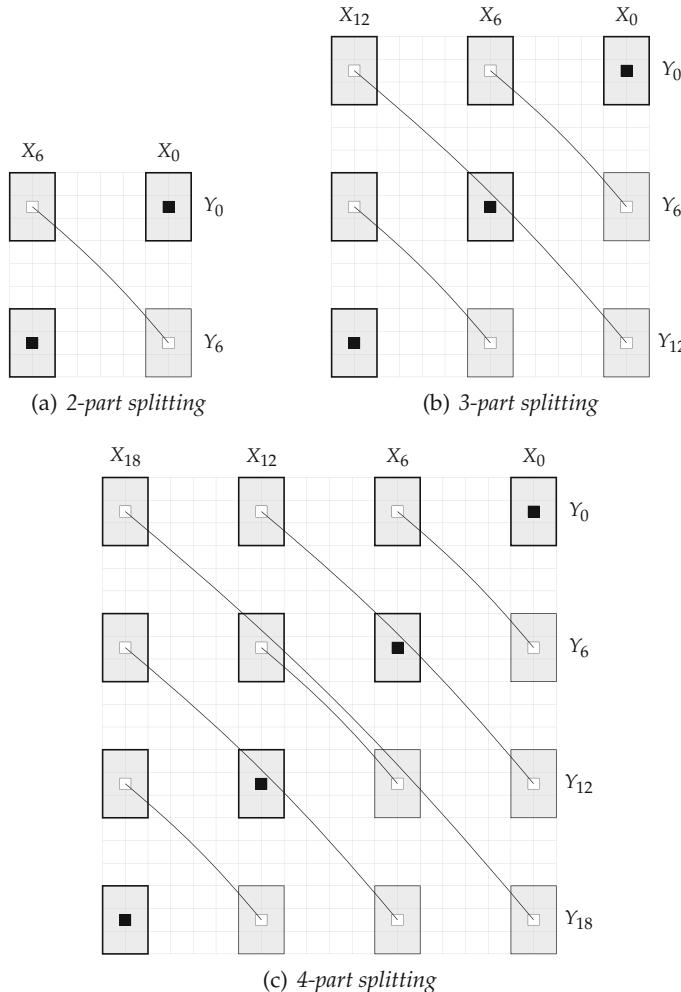


Fig. 8.29 Rectangular Karatsuba K-part splittings, represented as a tile.

Figure 8.30 shows several examples how this can be applied. The 64×72 is the smallest multiplier where Karatsuba for rectangular tiles of the 16×24 configuration can be applied. Here, only one multiplier can be save. However, much more can be gained when going to larger sizes as Fig. 8.30b and c demonstrate. In Fig. 8.30b, the 2-part splitting is applied several times in a shifted manner. The tiling in Fig. 8.30c is obtained from the 3-part splitting where the gaps are filled with the 2-part splitting pattern where possible.

In general, the tiling methods presented in Sect. 8.4.3 are easy to extend for these Karatsuba tiles [BKK20]. The properties regarding tile area, costs,

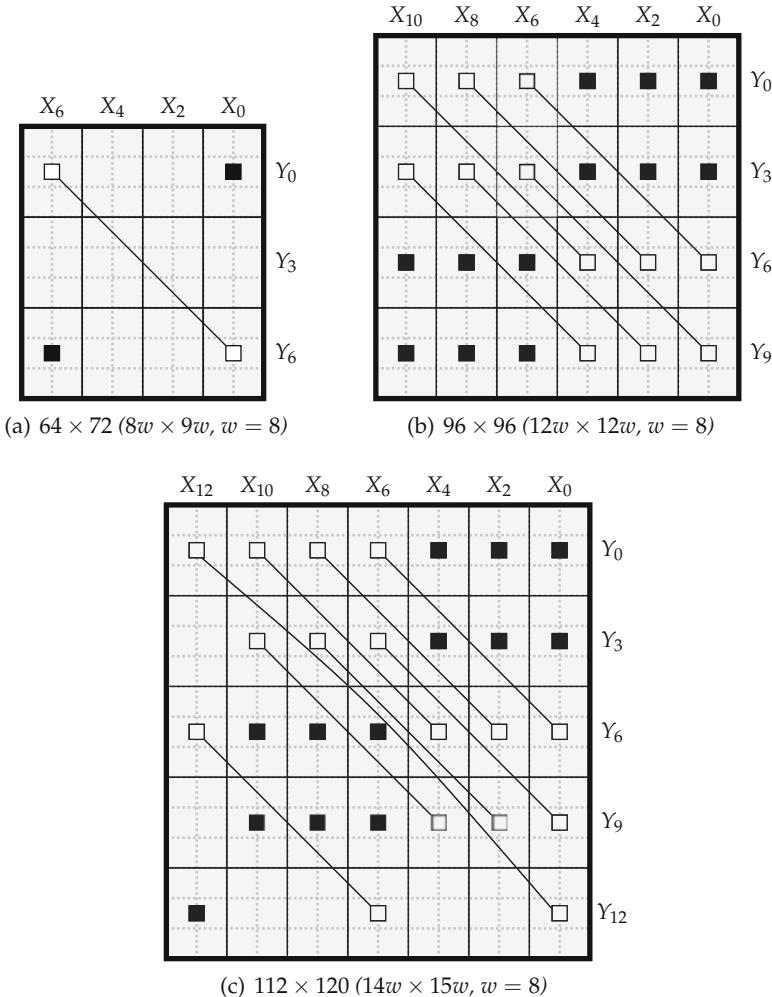


Fig. 8.30 Selected rectangular Karatsuba tilings. They use 16×24 tiles, mostly replicating the pattern in Fig. 8.28a.

and efficiencies are given in Table 8.5 for 2-, 3-, and 4-part splitting for both the classic Karatsuba using the DSPs as 17×17 square multipliers and the rectangular one using 16×24 multipliers. One can observe that using the rectangular multiplier results in a much better efficiency (both for LUTs and DSPs) compared to the square multipliers. Compared to a single DSP, Karatsuba with rectangular splittings increases the DSP efficiency at the cost of a slightly reduced LUT efficiency (due to the additional adders).

Table 8.5 Properties of Karatsuba tiles. This table extends Table 8.3 (the standard 24×17 tile is given again for comparison).

Shape	Tile area	$\text{cost}_s^{\text{LUT}}$	$\text{cost}_s^{\text{DSP}}$	E_s^{LUT}	E_s^{DSP}
24×17	408	22.55	1	18.13	408
2-part square split (17×17)	1156	110.5	3	10.46	385.3
3-part square split (17×17)	2601	256.7	6	10.13	433.5
4-part square split (17×17)	4624	476	10	9.71	462.4
2-part rect. split (16×24)	1536	126	3	12.19	512
3-part rect. split (16×24)	3456	312	6	11.08	576
4-part rect. split (16×24)	6144	580	10	11.21	614.4

Table 8.6 Operation counts for similar multiplier sizes.

Square tiles						Rectangular tiles						
Size	Karatsuba			Size	Tiling		Karatsuba					
	Mult	Pre-add	Post-add		Mult = Post-add	Mult	Pre-add	Post-add				
51×51	6	6	6	48×48	6	6	0	6				
68×68	10	12	22	64×72	12	11	2	13				
102×102	21	30	51	96×96	24	18	5	30				
119×119	28	42	70	112×120	35	27	7	43				

Table 8.6 compares different multiplier sizes using the following:

- the classic (square) Karatsuba formula using square 17×17 tiles,
- the conventional tiling (without any reduction) using rectangular (16×24) tiles,²
- Karatsuba's method extended to the rectangular tiles just discussed.

The comparison includes the operation counts in terms of the number of sub-multipliers, pre-adders, and post-adders used for all three methods. The sizes of the large multipliers were selected to be as close to each other as possible to achieve a direct comparison. The resulting tilings are shown in Fig. 8.30. Karatsuba's method using rectangular tiles requires the least multipliers except in the 68×68 case, where classic Karatsuba uses one sub-multiplier less. In terms of adders, the proposed method also significantly improves the numbers of pre- and post-adders. The main reason for this is of course the larger tile size, which leads to fewer sub-products, hence pre-adders.

² Note that 17×24 would be possible in this case, but this does not change the operation count we are considering here.

Note that for the larger multipliers using the proposed method, many of the pre-adders can be shared as they compute the same values. For the 96×96 multiplier, only 5 out of 12 pre-additions compute different values. For the 112×120 multiplier, only 7 pre-additions are required out of 16 (see [Kum+18]). For traditional Karatsuba, no such sharing is possible as each pre-addition only appears once.

References

- [A005126] A005126. On-Line Encyclopedia of Integer Sequences. <http://oeis.org/A005126>. 2010 (cit. on p. 238).
- [Ban+10] Sebastian Banescu, Florent de Dinechin, Bogdan Pasca, and Radu Tudoran. “Multipliers for floating-point double precision and beyond on FPGAs”. In: *ACM SIGARCH Computer Architecture News* 38 (2010), pp. 73–79 (cit. on pp. 10, 234, 238, 239).
- [Bea+02] Olivier Beaumont, Vincent Boudet, Fabrice Rastello, and Yves Robert. “Partitioning a Square into Rectangles: NPCompleteness and Approximation Algorithms”. In: *Algorithmica* 34.3 (2002), pp. 217–239 (cit. on p. 234).
- [Bew94] Gary W. Bewick. “Fast Multiplication: Algorithms and Implementation”. PhD thesis. Stanford University, 1994 (cit. on p. 222).
- [BKD21] Böttcher, Andreas and Kumm, Martin and de Dinechin, Florent. “Resource Optimal Truncated Multipliers for FPGAs”. In: *Symposium on Computer Arithmetic (ARITH)*, pp. 102–109. IEEE, 2021. (cit. on p. 239).
- [BKK20] Andreas Böttcher, Keanu Kullmann, and Martin Kumm. “Heuristics for the Design of Large Multipliers for FPGAs”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2020 (cit. on pp. 234, 236, 249, 252).
- [Boo51] Andrew D. Booth. “A Signed Binary Multiplication Technique”. In: *The Quarterly Journal of Mechanics and Applied Mathematics* (1951), pp. 236–240 (cit. on p. 217).
- [BW73] Charles R. Baugh and Bruce A. Wooley. “A Two’s Complement Parallel Array Multiplication Algorithm”. In: *IEEE Transactions on Computers* C-22.12 (1973), pp. 1045–1047 (cit. on p. 231).
- [BZ10] Richard P. Brent and Paul Zimmermann. *Modern Computer Arithmetic*. Vol. 18. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2010, p. 221 (cit. on p. 247).

- [DD07a] Jérémie Detrey and Florent de Dinechin. "Floating-Point Trigonometric Functions for FPGAs". In: *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2007, pp. 29–34 (cit. on pp. 212, 324).
- [DDP07] Jérémie Detrey, Florent de Dinechin, and Xavier Pujol. "Return of the hardware floating-point elementary function". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2007, pp. 161–168 (cit. on p. 212).
- [DP09] Florent de Dinechin and Bogdan Pasca. "Large multipliers with fewer DSP blocks". In: *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2009, pp. 250–255 (cit. on pp. 222, 231, 234).
- [DRC14] Theo A. Drane, Thomas M. Rose, and George A. Constantinides. "On the Systematic Creation of Faithfully Rounded Truncated Multipliers and Arrays". In: *IEEE Transactions on Computers* 63.10 (2014), pp. 2513–2525 (cit. on p. 238).
- [EL04] Miloš D. Ercegovac and Tomás Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004 (cit. on pp. 3, 11, 23, 214, 218, 259, 267, 303).
- [Gao+12] S. Gao, D. Al-Khalili, N. Chabini, and P. Langlois. "Asymmetric Large Size Multipliers with Optimised FPGA Resource Utilisation". In: *Computers & Digital Techniques, IET* 6.6 (2012), pp. 372–383 (cit. on p. 222).
- [GMPweb] *The GNU Multiple Precision Arithmetic Library*. URL: <http://gmplib.org/> (cit. on p. 247).
- [Kak+16] Ahmet Kakacak, Aydin Emre Guzel, Ozan Cihangir, Sezer Gören, and H. Fatih Uğurdağ. "Fast Multiplier Generator for FPGAs with LUT based Partial Product Generation and Column/Row Compression". In: *Integration, the VLSI Journal, Elsevier* (2016), pp. 147–157 (cit. on p. 230).
- [KAZ15] Martin Kumm, Shahid Abbas, and Peter Zipf. "An Efficient Softcore Multiplier Architecture for Xilinx FPGAs". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2015, pp. 18–25 (cit. on pp. 214, 230, 231).
- [KH11] Hou-Jen Ko and Shen-Fu Hsiao. "Design and Application of Faithfully Rounded and Truncated Multipliers With Combined Deletion, Reduction, Truncation, and Rounding". In: *Transactions on Circuits and Systems II: Express Briefs* 58.5 (2011), pp. 304–308 (cit. on p. 238).
- [KO63] Anatoly Karatsuba and Yuri Ofman. "Multiplication of Multidigit Numbers on Automata". In: *Soviet Physics Doklady* 7 (1963), p. 595 (cit. on pp. 244, 247).
- [Kor02] Israel Koren. *Computer Arithmetic Algorithms*. 2nd ed. Prentice-Hall, 2002 (cit. on pp. 23, 211).

- [KS97] Eric J. King and Earl E. Swartzlander. "Data-Dependent Truncation Scheme for Parallel Multipliers". In: *Asilomar Conference on Signals, Circuits and Systems*. Vol. 2. IEEE, 1997, pp. 1178–1182 (cit. on p. [238](#)).
- [Kum+17] Martin Kumm, Johannes Kappauf, Matei Istoan, and Peter Zipf. "Optimal Design of Large Multipliers for FPGAs". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2017, pp. 131–138 (cit. on pp. [222](#), [234](#), [236](#)).
- [Kum+18] Martin Kumm, Oscar Gustafsson, Florent de Dinechin, Johannes Kappauf, and Peter Zipf. "Karatsuba with Rectangular Multipliers for FPGAs". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2018 (cit. on pp. [246](#), [249](#), [250](#), [255](#)).
- [Lan+19] Martin Langhammer, Bogdan Pasca, Gregg Baeckler, and Sergey Gribok. "Extracting INT8 Multipliers from INT18 Multipliers". In: *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2019, pp. 114–120 (cit. on pp. [222](#), [223](#), [226](#)).
- [LB18] Martin Langhammer and Gregg Baeckler. "High Density and Performance Multiplication for FPGA". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2018 (cit. on p. [222](#)).
- [LBG19] Martin Langhammer, Gregg Baeckler, and Sergey Gribok. "Fractal Synthesis". In: *International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM, 2019, pp. 202–211 (cit. on p. [222](#)).
- [Lev84] Leonid A. Levin. "Problems, Complete in "Average" Instance". In: *ACM Symposium on Theory of Computing*. 1984, p. 465 (cit. on p. [234](#)).
- [Lev87] Leonid A. Levin. "Average Case Complete Problems". In: *SIAM Journal on Computing* 15.1 (1987), pp. 285–286 (cit. on p. [234](#)).
- [Mon05] Peter L. Montgomery. "Five, Six, and Seven-Term Karatsuba-Like Formulae". In: *IEEE Transactions on Computers* 54.3 (2005), pp. 362–369 (cit. on p. [246](#)).
- [MTV06] Romain Michard, Arnaud Tisserand, and Nicolas Veyrat-Charvillon. "Carry Prediction and Selection for Truncated Multiplication". In: *Workshop on Signal Processing Systems (SiPS)*. IEEE, 2006, pp. 339–344 (cit. on p. [238](#)).
- [Mul16] Jean-Michel Muller. *Elementary Functions, Algorithms and Implementation*. 3rd ed. Birkhäuser Boston, 2016 (cit. on pp. [24](#), [212](#)).
- [Ng92] Kwok C. Ng. *Argument reduction for huge arguments: good to the last bit*. Technical Report. SunPro, 1992 (cit. on p. [212](#)).
- [Pas12] Bogdan Pasca. "High-Performance Floating-Point Computing on Reconfigurable Circuits". PhD thesis. École Normale Supérieure de Lyon, 2012 (cit. on pp. [222](#), [246](#)).

- [Pet+10] Nicola Petra, Davide De Caro, Valeria Garofalo, Ettore Napoli, and Antonio G.M. Strollo. "Truncated Binary Multipliers With Variable Correction and Minimum Mean Square Error". In: *Transactions on Circuits and Systems I: Regular Papers* 57.6 (2010), pp. 1312–1325 (cit. on p. [238](#)).
- [PI11] Hadi Parandeh-Afshar and Paolo Ienne. "Measuring and Reducing the Performance Gap between Embedded and Soft Multipliers on FPGAs". In: *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2011, pp. 225–231 (cit. on p. [231](#)).
- [SP92] Paul E. Sweeney and Elizabeth Ridenour Paternoster. "Cutting and Packing Problems: A Categorized, Application-Orientated Research Bibliography". In: *Journal of the Operational Research Society* 43.7 (1992), pp. 691–706 (cit. on p. [234](#)).
- [SS93] Michael J. Schulte and Earl E. Swartzlander. "Truncated Multiplication with Correction Constant". In: *IEEE Workshop on VLSI Signal Processing*. 1993, pp. 388–396 (cit. on pp. [238, 241](#)).
- [Wal14] E. Georges Walters III. "Partial-Product Generation and Addition for Multiplication in FPGAs with 6-Input LUTs". In: *Asilomar Conference on Signals, Circuits and Systems*. IEEE, 2014, pp. 1247–1251 (cit. on p. [231](#)).
- [Wal16] E. Georges Walters III. "Array Multipliers for High Throughput in Xilinx FPGAs with 6-Input LUTs". In: *Computers, MDPI* 5.4 (2016), pp. 1–25 (cit. on pp. [214, 231](#)).
- [Xil11] *Virtex-6 FPGA DSP48E1 Slice User Guide UG369 (v1.3)*. Xilinx Inc. 2011 (cit. on p. [231](#)).
- [Xil17] Yao Fu, Ephrem Wu, Ashish Sirasao, Sedny Attia, Kamran Khan, and Ralph Wittig. *Deep Learning with INT8 Optimization on Xilinx Devices (White Paper)*. Tech. rep. Xilinx, Inc., 2017 (cit. on pp. [222, 223, 226](#)).
- [Xil20] *Convolutional Neural Network with INT4 Optimization on Xilinx Devices (White Paper, WP521)*. Xilinx Inc. 2020 (cit. on p. [226](#)).



9

CHAPTER 9

Fixed-Point Division

Black holes are where God divided by zero.

Steven Wright

In typical applications, division is not as frequent as addition or multiplication (Oberman and Flynn (IEEE Trans Comput 46.2, 154–161, 1997)), but its hardware implementation has nevertheless received considerable attention. Dividers may be implemented in a variety of ways, with two main families each declined in countless variations. In this rich landscape, there is a divider for each context, but finding the right one may be a daunting task. This chapter provides a bird’s-eye view on this landscape and a focus on a few detailed implementations.

The subject of division is very well covered in specialized textbooks [EL94; EL04]. There are two main families of division algorithms [OF97b; EL04] which can be used to divide a number X by a number D , namely, *digit-recurrence* and *multiplicative* methods.

The *digit-recurrence* family [EL94], to which the decimal paper-and-pencil technique belongs, is presented in Sect. 9.2. It computes a quotient Q digit by digit and simultaneously computes a remainder R such that $X = QD + R$, with some constraint on R , for instance, $0 \leq R < D$. This computation is iterative, and each iteration is relatively simple (based on additions and digit-by-number multiplications), but the number of iterations is proportional to the result precision. In this family, the iteration is also usually exact: R is computed exactly. Rounding Q is then simple.

The second family uses *multiplicative techniques* to compute division. A first idea, reviewed in Sect. 9.3, is to use an approximation to the inverse $1/D$ which is multiplied by X to obtain an approximation to the quotient X/D .

The approximation to $1/D$ may use plain tabulation for small precisions. For larger precisions it is natural to also consider multiplication-based reciprocal approximation techniques [Obe99; Mar00; PB02; Jai+14; IP17]. Using a series expansion of $1/D$ enables to merge the multiplication by X in the evaluation of this series: this technique is detailed in Sect. 9.4.

Multiplicative methods have been developed for microprocessors with multiplier hardware, in order to exploit this resource to compute division [OF97a; OF97b; SL97; Obe99; Mar00]. Later they also have been used on FPGAs with embedded multipliers [WBL06; Pas12; Jai+14; IP17]. Multiplication-based iterations typically are more expensive than digit-recurrence ones, but converge faster. Rounding in multiplicative techniques is also slightly more complicated than in digit recurrences, due to the approximations involved which require a careful error analysis.

A peculiarity of the division operation is that the division of (normalized) floating-point numbers is typically simpler or cheaper than the division of fixed-point numbers. One reason is that it avoids the singularity of dividing by a number close to zero. In other terms, where floating-point addition and multiplication are naturally built on top of fixed-point addition and multiplication, for division it is almost the other way around. Our reader should therefore not be surprised that some fixed-point division techniques reviewed in this chapter have constraints on their inputs that correspond to normalized floating-point significands.

9.1 Fixed-Point Division: Problem Formulation

Let us first reconcile the various points of view that may exist on division. Given two integers X and $D \neq 0$, Euclidean division consists of finding two integers Q (the quotient) and R (the remainder) such that

$$X = QD + R \quad \text{with} \quad 0 \leq R < D. \quad (9.1)$$

In this formulation X and D may be natural integers or signed ones. When they are signed, the standard convention is that the quotient Q is signed as well, and the constraint on the remainder becomes $0 \leq R < |D|$. Note that this entails that Q is always $\left\lfloor \frac{X}{D} \right\rfloor$, rounded *down* (toward $-\infty$). This convention is supported by most programming languages when dividing signed integers. For instance, typing in a Python console $4/3$ returns 1 , while typing $-4/3$ returns -2 .

However, this is just a convention: the constraint on R can be replaced by any other constraint that ensures unicity. For instance, the interval of possible values for R can be centered around zero, e.g., $-\lfloor D/2 \rfloor \leq R < \lceil D/2 \rceil$.

Euclidean division can be rewritten

$$\frac{X}{D} = Q + \frac{R}{D} \quad \text{with} \quad 0 \leq \frac{R}{D} < 1 \quad (9.2)$$

or, noting $\delta = R/D$ and with a unicity constraint that centers the remainder around zero,

$$\frac{X}{D} = Q + \delta \quad \text{with} \quad -1/2 \leq \delta < 1/2. \quad (9.3)$$

The essence of the unicity constraint is that 1 is the unit in the last place (ulp) of the quotient Q . To generalize to a fixed-point quotient Q , all we need is to define u_Q , the ulp of the format of Q (see Sect. 3.1.2), and we obtain the most general formulation

$$\frac{X}{D} = Q + \delta \quad \text{with} \quad -\frac{u_Q}{2} \leq \delta < \frac{u_Q}{2}. \quad (9.4)$$

This formulation is also that of a rounding to the nearest (with ties always rounded up): all this derivation shows that the remainder (in Euclidean division) and the rounding error (in the division of two reals) are two facets of the same coin. This equivalence may be exploited both ways:

- Using an algorithm that computes both quotient and remainder, such as the schoolbook algorithm, we will be able to decide the rounding to the nearest of the quotient.
- Using an algorithm that evaluates an approximation of the quotient, it is possible to compute the remainder as soon as the approximation is accurate enough.

In the second case, given an approximation Q of the exact real X/D , it is useful to define a generalization of the remainder as

$$R = X - QD \quad (9.5)$$

and the link with the absolute error δ of Q with respect to the exact quotient¹ is, as already defined,

$$\delta = \frac{X}{D} - Q = \frac{R}{D}. \quad (9.6)$$

We have discussed in Chap. 3 the benefits of relaxing rounding to the nearest to, for instance, faithful approximations. Similarly, it will be possible (and useful) to relax the constraint on the generalized remainder.

¹ This definition of δ is the opposite of an error as defined in Chap. 3: there, we advocated that the error should always be defined as the less accurate term (here Q) minus the more accurate one (here X/D). We choose to make an exception here, for consistency with the usual remainder definition. It will be of no practical significance, because error analysis is mostly concerned with absolute values of errors.

Note that so far, all that matters is the format of Q that defines the ulp u_Q . Let us now see the relationships that exist between the formats of X , D , Q , and R .

9.1.1 Format Considerations for Unsigned Integer Division

First consider the division of a w_X -bit unsigned integer X by a w_D -bit unsigned integer D . Excluding the special case of division by 0, the maximum value of the quotient Q happens when dividing by $D = 1$: the format of Q should be the format of X . The maximum value of the remainder is defined by the unicity constraint in (9.1). It entails that the format of R is that of D . We conclude that in the case of Euclidean division as defined by (9.1), we have $w_Q = w_X$ and $w_R = w_D$.

An interesting observation is that the result (Q, R) requires the same number of bits as the input (X, D) .

9.1.2 Format Considerations for Signed Integer Division

Let us now consider Euclidean division of *signed* integers in two's complement. If only X is signed and D is unsigned, all the previous remains true: the format of Q is that of X , and the format of R is that of D . However, as soon as D is signed (still with the constraint of (9.1)), there is a subtlety due to the asymmetry of two's complement: we can now divide by -1 the largest negative value -2^{w_X-1} , and the result is not representable on w_X bits in two's complement – we need $w_X + 1$ bits. Conversely, D can take values -2^{w_D-1} to $-2^{w_D-1} - 1$, but the unicity constraint in this case, $0 \leq R < |D|$ can be rewritten $0 \leq R < 2^{w_D-1}$: thus R fits on an unsigned integer with one bit less than D .

In summary, it remains true that the result (Q, R) requires the same number of bits as the input (X, D) , although the formats are no longer identical.

9.1.3 Format Considerations for Fixed-Point Division

In the most general case when X , D , Q , and R are all fixed-point numbers, each with its MSB m and LSB ℓ , we have eight parameters to consider.

In most cases, the relevant point of view is the error-centered formulation of (9.4). The maximal value of Q will be attained when dividing the maximum possible value of X by the minimum possible non-zero value of D (all in absolute value). This defines m_Q as $m_X - \ell_D$, plus or minus one depend-

ing on the respective signednesses – we deliberately keep it vague here for the sake of simplicity. As for ℓ_Q , it is more of a free parameter, controlling how accurate the quotient will be. However, as soon as this parameter is defined (with the ulp of Q being $u_Q = 2^{\ell_Q}$), the LSB of the general remainder is defined by (9.5) as $\ell_R = \ell_Q + \ell_D$, while the constraint

$$-\frac{u_Q D}{2} \leq \delta D = R < \frac{u_Q D}{2} \quad (9.7)$$

defines $m_R = \ell_Q + m_D$ (again plus minus one depending on the respective signedness of R and D). What is important is that the size in bits of R is that of D , whatever ℓ_Q .

This should ring a bell in our reader's mind of distant memories of the schoolbook algorithm for doing division. Indeed, this algorithm consists of incrementally computing Q and R with more and more accuracy (i. e., with decreasing ℓ_Q). At each step of this algorithm, an intermediate remainder is computed, and its size is always that of D , irrespective of the sizes of X and Q . We will review this algorithm and generalize it in Sect. 9.2. Before this, however, there is one last specific case that deserves special treatment.

9.1.4 Division of Normalized Floating-Point Significands

The division of floating-point numbers is studied in Sect. 11.3 of this book. It essentially consists of subtracting the exponents and dividing the significands. Floating-point normalization, rounding, and other floating-point-specific issues are addressed in Sect. 11.3, while this chapter addresses in depth the division of the normalized significands.

Normalized significands, in this book, are fixed-point numbers in a $\text{ufix}(0, -w_F)$ format with an additional specificity: their MSB is equal to 1; therefore, they belong to $[1, 2)$ instead of $[0, 2)$. The quotient of two normalized significands in $[1, 2)$ belongs to $(1/2, 2)$, and thus in this case $m_Q = 1$ instead of $m_X - \ell_D$.

The literature about floating-point division traditionally considers that the significands of normalized floating-point numbers belong to $[0.5, 1)$ instead of $[1, 2)$ as in this book. We will sometimes follow this convention in the present chapter, and the reader should be convinced that this is absolutely irrelevant, since the quotient of dividing X by D is equal to the quotient of dividing $2X$ by $2D$. For instance, the previous remark (the quotient of two normalized significands belongs to $(1/2, 2)$; hence $m_Q = 1$) holds for both significand conventions.

An important remark is that when dividing normalized significands, the denominator cannot come close to zero. This will in many cases simplify the division problem, and for this reason most of the literature about hardware dividers focuses on the division of normalized significands.

9.2 Digit-Recurrence Algorithms

Since this family of algorithms is a generalization of the schoolbook algorithm, Fig. 9.1 is an example of decimal division using the algorithm learned at school (and probably forgotten since then) for dividend $X = 12,230$ and divisor $D = 453$.

This schoolbook algorithm is iterative. Each iteration first determines a quotient digit q_i . This step will be termed *quotient digit selection* in the sequel. Each iteration then uses this quotient digit to compute a partial remainder (3170 in Fig. 9.1b) that will play the part of the initial dividend for the next iteration. This computation itself consists of two operations: first compute the product $q_i D$, a simpler case of multiplication since it multiplies a digit by a number (its result is in italic in Fig. 9.1); then subtract $q_i D$, suitably shifted, from the previous remainder to get the next one. From now on, we may consider that iteration i computes the next remainder R_{i+1} out of the current remainder R_i , and the dividend X is only used to define the initial remainder $R_0 = X$. These notations and others that will be used throughout this section are summarized in Table 9.1.

Both multiplication and subtraction are relatively straightforward, although many techniques (reviewed in the sequel) have been designed to accelerate them.

What is really tricky is the first step, the quotient digit selection. To illustrate this, the example of Fig. 9.1 was constructed as a particularly difficult case. We learned at school to try to deduce the quotient digit from the observation of the first few digits of remainder and divisor, but it is normal to get it wrong sometimes: it happens when the current remainder is close to a multiple of the divisor (in our example, 3170 is close to $7 \times 453 = 3171$). Consider that if the current remainder had been 3172, then the correct quotient digit would indeed have been 7. Obviously, these two situations cannot be discriminated by observing only the leftmost digits of 3170...

A helpful remark is that, when we get the quotient digit wrong (either way), it is wrong by one unit at most.

Table 9.1 Common notations used in this section.

Symbol	Meaning
β	The base or radix
α	When β is a power of two, $\beta = 2^\alpha$
γ	Digit-set parameter: digits belong to $\{-\gamma, \dots, 0, \dots, \gamma\}$
ρ	The redundancy factor, $\rho = \frac{\gamma}{\beta-1}$
q_j	A radix- β digit of the quotient
Q_i	The partial quotient at iteration i
R_i	The partial remainder at iteration i

$$\begin{array}{r} 12230 \\ \hline 453 \\ - 2 \\ \hline = 3170 \end{array} \qquad \begin{array}{r} 12230 \\ \hline 453 \\ - 9060 \\ = 3170 \\ - 3171 \\ \hline \end{array}$$

(a) First digit is easy

$$\begin{array}{r} 12230 \\ \hline 453 \\ - 9060 \\ = 3170 \\ - 2718 \\ \hline = 452 \end{array}$$

(b) Second not so

$$\begin{array}{r} 12230 \\ \hline 453 \\ - 9060 \\ = 3170 \\ - 3171 \\ \hline \end{array}$$

(c) Try 7 and check

$$\begin{array}{r} 12230 \\ \hline 453 \\ - 9060 \\ = 3170 \\ - 2718 \\ \hline = 452 \end{array}$$

(d) Wrong! it was 6

$$\begin{array}{r} 31711 \\ - 2718 \\ \hline = 4531 \end{array}$$

(e) Another way of getting it wrong

Fig. 9.1 A step-by-step example of division in decimal.

It is also important to remark that the difficulty comes from the strict constraint that we impose on the schoolbook algorithm: the next remainder should be positive (hence the backtracking from 7 to 6 in Fig. 9.1d). It should be strictly less than the divisor; otherwise, as illustrated in Fig. 9.1e for 31,711 divided by 453, the next iteration would have to select a quotient digit larger than 9, which it can't. In short, it is the strict constraint on the generalized remainder defined in Sect. 9.1 that ensures the convergence of the schoolbook algorithm, but it also ensures that there will be cases where approximate quotient digit selection (as in “consider only a few leftmost digits”) will fail.

The remainder of this section exposes a family of division algorithms that generalize the schoolbook algorithm while providing various solutions to the difficult quotient digit selection problem, mostly by relaxing the constraint on the remainder. Before this, let us start with a straightforward transposition of the schoolbook algorithm to binary.

9.2.1 Schoolbook Integer Division in Binary (Restoring Division)

In binary, we have only two possible values for a quotient digit: 0 or 1. Therefore, the product $q_i D$ can be either 0 or D . The iteration can thus be simplified as follows: assume that $q_i = 1$ and tentatively subtract D (suitably shifted). If the result is negative, it means that q_i was 0: ignore the result of the subtraction; the next remainder is the previous one. If the result of the subtraction is positive, it is the next remainder.

We invite our reader to use this schoolbook method to attempt to divide 100011 by 111 (that is 35 divided by 7). The answer is given in Fig. 9.2a, where trial subtractions are shown in brackets.

i	R_i	$R_i - 2^{w_X-1-i}D$	q_{w_Q-i-1}
0	35	$35 - 2^5 \cdot 7 = -189$	0
1	35	$35 - 2^4 \cdot 7 = -77$	0
2	35	$35 - 2^3 \cdot 7 = -21$	0
3	7	$35 - 2^2 \cdot 7 = 7$	1
4	7	$7 - 2^1 \cdot 7 = -7$	0
5	0	$7 - 2^0 \cdot 7 = 0$	1

(a) schoolbook view

(b) step-by-step recurrence

Fig. 9.2 An example of schoolbook binary division of integers, computing 35 : 7.

The schoolbook method can be formalized using the recurrence

$$R_0 = X \tag{9.8}$$

$$q_{w_X-1-i} = \begin{cases} 1 & \text{when } R_i - 2^{w_X-1-i}D \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{9.9}$$

$$R_{i+1} = R_i - 2^{w_X-1-i}q_{w_X-1-i}D \quad . \tag{9.10}$$

Figure 9.2b shows the intermediate values in each iteration i using $w_X = 6$ bit, leading to $Q = 000101$.

As a consequence the algorithm must perform six iterations (the first three of which, outputting zeroes, can be skipped when the computation is done by hand).

In (9.10) the shift of $q_{w_X-1-i}D$ with respect to R_i depends on the iteration index i . A constant shift will be more hardware-friendly and can be achieved by a change of variable $R_i = 2^{w_X-i}R'_i$. The recurrence becomes

$$R'_0 = 2^{-w_X} X \quad (9.11)$$

$$q_{w_X-1-i} = \begin{cases} 1 & \text{when } 2R'_i - D \geq 0 \\ 0 & \text{otherwise} \end{cases} \quad (9.12)$$

$$R'_{i+1} = 2R'_i - q_{w_X-1-i}D \quad \forall i \geq 0. \quad (9.13)$$

Figure 9.3 shows the intermediate steps for the example of 35 : 7.

With (9.13) no longer involving an iteration-dependent shift, all the iterations become identical to the last one.

This very basic algorithm is often called “restoring division” for historical reasons dating back to heroic implementations on register-constrained processors.²

From the initialization $R'_0 = 2^{-w_X} X$, it is clear that the word size of R_0 is $w_R = w_X + w_D$. The subtraction $2R'_i - q_{w_X-1-i}D$ only affects the integer bits of R'_i , since the fractional bits of D are zeroes (see Fig. 9.3). Hence, the actual subtractor only requires $w_D + 1$ bits (where the +1 is for the sign bit). The LSBs of the difference are the fractional bits of $2R'_i$. In Fig. 9.3, the $w_D + 1$ bits that actually have to be subtracted by hardware are shown in blue, and the w_X bits that are just replicated from $2R'_i$ (or, by recurrence, from the input X) are shown in green.

It can be seen that the iteration subtraction uses one new bit of X at each iteration: iteration i consumes x_{w_X-1-i} , the bit of weight $w_X - 1 - i$ of X . This is very similar to the decimal paper-and-pencil division when $w_X > w_D$, where each step considers one new decimal digit of X .

To sum up, the iteration hardware is shown in Fig. 9.4. It inputs the integer part of R'_i (as an unsigned integer) and one bit of X , and outputs the integer part of R'_{i+1} and one bit of Q . The multiplexer always selects an input whose sign bit is zero. This is obvious when the sign bit of the difference is 0. When the sign bit of the difference is 1, the multiplexer selects $2R'_i$, but then $0 \leq 2R'_i < D$ and D has been left-extended with a 0 sign bit, so the sign bit of $2R'_i$ is also a 0. This sign bit can therefore be discarded for the next iteration, so $[R'_{i+1}]$ is again a w_D -bit unsigned integer.

The latency of an architecture using Fig. 9.4 is w_Q clock cycles, where the clock cycle is constrained by the delay of one subtraction of w_D bits. If one of these architectures is unrolled in space, the delay of one division is not improved: the multiplexer is controlled by the sign bit (the MSB) of the result of the subtraction, which comes last due to carry propagation. Unrolling in space may improve the throughput thanks to pipelining, though.

² In a microprocessor where you have only one register to hold your remainder, then when $q_i = 0$, you must “restore” its value by adding back D , in which case the iteration costs two additions. A variant called non-restoring division [EL04; Par10] was developed to ensure exactly one addition or subtraction at each iteration, but at the expense of one more iteration to correct a possibly negative final remainder. When targeting ASICs, there is no benefit in the non-restoring version. It does map better to FPGAs with small LUTs, but the high-radix dividers studied in the sequel map even better to modern FPGAs.

i	R'_i	$2R'_i - D$	q_{w_X-i-1}
		0001.000110	
0	0 000.100011	+ 1001.000000	0
		= 1010.000110 < 0	
		0010.001100	
1	1 001.000110	+ 1001.000000	0
		= 1011.001100 < 0	
		0100.011000	
2	2 010.001100	+ 1001.000000	0
		= 1101.011000 < 0	
		1000.110000	
3	3 100.011000	+ 1001.000000	1
		= 0001.110000 > 0	
		0011.100000	
4	4 001.110000	+ 1001.000000	0
		= 1100.100000 < 0	
		0111.000000	
5	5 011.100000	+ 1001.000000	1
		= 0000.000000 = 0	

Fig. 9.3 Binary computations using the binary restoring algorithm, with $w_X = 6$ and $w_D = 3$, computing $35 : 7$. The subtractions of D are shown as the addition of its two's complement representation $-D = 1001$.

It is worth noting that the recursion can proceed to compute an arbitrary number of fractional digits of the quotient (just as in the paper-and-pencil decimal division). In this case the architecture must input zeroes for all the fractional bits of X , or formally $x_{w_X-1-i} = 0$ when $w_X - 1 - i < 0$.

What we have so far is therefore a low-cost but long-latency divider. Let us now address higher-performance architectures. For that, we will first generalize the division operation to higher radix.

9.2.2 General Radix- β Formalization of Integer Division

We have not yet formally proven that the recurrence actually computes the quotient and remainder. Let us do this in the general case of radix β (or *base*, another name for the radix). We have considered $\beta = 10$ for the decimal paper-and-pencil algorithm and $\beta = 2$ for binary. This generalization to arbitrary radices will be useful, in particular with radices that are small

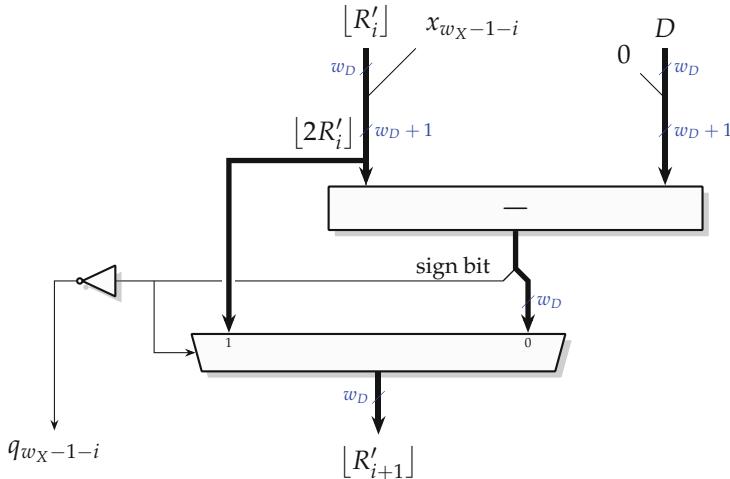


Fig. 9.4 Hardware for one step of restoring division.

powers of two ($4, 8, 16, 32, \dots$). Indeed, as seen in Sect. 2.3, a binary (radix-2) number can be interpreted as a radix- 2^α one at no cost, simply by grouping its bits in chunks of α consecutive bits. Hexadecimal is a well-known example when $\alpha = 4$: each group of 4 bits can take the values $\{0, \dots, 15\}$ and can therefore be interpreted as a radix-16 digit.

More generally, a w_X -bit binary integer $X = \sum_{i=0}^{w_X-1} 2^i x_i$ can be rewritten in radix $\beta = 2^\alpha$ as k digits, where $k = \lceil \frac{w_X}{\alpha} \rceil$, as follows:

$$X = \sum_{i=0}^{k-1} X_i \cdot (2^\alpha)^i, \text{ where } X_i = \sum_{j=0}^{\alpha-1} 2^j x_{j+\alpha i}, X_i \in \{0, \dots, 2^\alpha - 1\}. \quad (9.14)$$

The advantage of using a higher radix for division is that the number of iterations is equal to the number of digits: a hexadecimal division algorithm will require 4 times fewer iterations than the plain binary algorithm. Of course, each iteration will be more expensive: the art of divider design consists of finding the proper balance between number of iterations and cost of each iteration.

We can generalize the restoring division defined in (9.12) and (9.13) from radix 2 to radix β by replacing 2 by β . If we leave the digit selection problem for later, the recurrence that defines the core of an integer division iteration is

$$R_0 = \beta^{-k} X \quad (9.15)$$

$$R_{i+1} = \beta R_i - q_{k-i-1} D \quad \forall i \geq 0 \quad (9.16)$$

where

- The multiplication by β will come at no cost: it is a shift left by one digit (or α bits in radix $\beta = 2^\alpha$).
- q_{k-i-1} is one digit in radix β (between 0 and 9 in decimal, between 0 and 1 in binary, between 0 and $2^\alpha - 1$ in radix $\beta = 2^\alpha$, but we are going to study fancier digit sets in the sequel).

Unrolling i iterations, we get

$$R_i = \beta R_{i-1} - q_{k-i} D \quad (9.17)$$

$$= \beta^2 R_{i-2} - \beta q_{k-i+1} D - q_{k-i} D \quad (9.18)$$

$$= \beta^3 R_{i-3} - (\beta^2 q_{k-i+2} + \beta q_{k-i+1} + q_{k-i}) D \quad (9.19)$$

...

$$= \beta^i R_0 - \left(\underbrace{\sum_{j=0}^{i-1} \beta^j q_{k-i+j}}_{=Q} \right) D \quad (9.20)$$

After processing $i = k$ iterations and setting $R_0 = X\beta^{-k}$, we get

$$R_k = X - \underbrace{\left(\sum_{j=0}^{k-1} \beta^j q_j \right)}_{=Q} D, \quad (9.21)$$

which can be rewritten

$$X = QD + R_k = QD + R. \quad (9.22)$$

For two integers X and D , this is the definition of the Euclidean division (9.1) if we add the remainder constraint from (9.1):

$$0 \leq R = R_k < D. \quad (9.23)$$

It is natural to impose on all the R_i the constraint we expect on the remainder R_k :

$$0 \leq R_i < D. \quad (9.24)$$

We also need to prove that quotient digit selection is possible at each step. These two questions are linked: let us prove by induction that if we have $0 \leq R_i < D$ at iteration i , then it is possible to select a digit q_{k-i-1} to ensure that $0 \leq R_{i+1} < D$.

This is straightforward: our experience with paper-and-pencil division suggests to define

$$q_{k-i-1} = \left\lfloor \frac{\beta R_i}{D} \right\rfloor. \quad (9.25)$$

The induction hypothesis $0 \leq R_i < D$ ensures that $q_i \in \{0, \dots, \beta - 1\}$: thus q_{k-i-1} is indeed a radix- β digit. We also have $0 \leq \frac{\beta R_i}{D} - q_{k-i-1} < 1$ by definition of the floor function; hence $0 \leq \beta R_i - q_{k-i-1} D = R_{i+1} < D$.

Of course, to ensure a converging division algorithm, we also need to choose an initial alignment such that $0 \leq R_0 < D$. This is the case for the integer division problem with the initial condition $R_0 = X/\beta^k$: as $X < \beta^k$ by definition, we have $R_0 < 1 \leq D$.

9.2.3 General Radix- β Formalization for the Division of Normalized Significands

As already remarked in Sect. 9.1.4, p. 263, the quotient of two normalized significands belongs to $(1/2, 2)$ and will therefore be written in radix β as $Q = q_0.q_{-1}q_{-2}q_{-3}\dots$, the digits still being computed most significand first: in this case iteration i determines q_{-i} and computes

$$R_{i+1} = \beta R_i - q_{-i} D. \quad (9.26)$$

The initial R_i should use X , suitably scaled: the convergence requires that $0 \leq R_i < D$ for all i . With $X \in [1, 2)$ and $D \in [1, 2)$, we have $X/2 < 1 < D$, so $X/2$ is a valid initial remainder (the same conclusion is attained by considering both X and D in $[1/2, 1)$). By using $X/2$ as initial remainder, we now divide $X/2$ by D , but this can be corrected by adding 1 to the exponent of the quotient. Also, this implies $Q < 1$ so $Q = 0.q_{-1}q_{-2}q_{-3}\dots q_{-k}$: the algorithm starts with $R_1 = X/2$ and iterates (9.26) from $i = 1$ to $i = k$.

Finally, for a floating-point format with w_F fraction bits, the fixed-point format of all the R_i is that of $R_1 = X/2$: it is a ufix($-1, -w_F - 1$). For a correctly rounded quotient, we must compute $w_F + 3$ bits of Q (see Sect. 11.3), so the number of iterations is $k = \left\lceil \frac{w_F + 3}{\alpha} \right\rceil$.

The whole proof of the previous section then applies to significand division, after suitable scaling by powers of β .

In the following we focus on this division of normalized significands, since it allows for simpler notations as iteration i computes quotient digit q_{-i} .

9.2.4 Using Redundant Digit Sets

As the proof of the division algorithm was tight, the choice of q_{-i} defined by (9.25) was unique. Besides, the computation of q_{-i} in (9.25) is itself a division and is therefore barely simpler than the full division of X by D : again see Fig. 9.1.

The classic solution to this problem is to use a different, larger digit set for the q_{-i} . Such digit sets were introduced in Sect. 2.4, p. 46. Division algorithms based on them are collectively termed SRT algorithms after Sweeney, Robertson [Rob58], and Tocher [Toc58] who introduced them independently in 1958.

Redundancy will be the key to easing the quotient digit selection. Remember that the difficult digit selection situations happen when, by looking only at the leading digits of R_i and D , we hesitate between two values for the next quotient digit q_{-i} , for instance, $q_{-i} = 2$ and $q_{-i} = 3$. Intuitively, with a redundant digit set, committing to either value will be less dramatic, since a wrong choice can be compensated in later iterations.

Consider, for instance, decimal division ($\beta = 10$) with the digit set $\{\bar{7}, \bar{6}, \dots, \bar{1}, 0, 1, \dots, 7\}$. With this digit set, $q_{-i} = 2$ can be followed by $q_{-i-1} = 7$, and 2.7 is closer to 3 than to 2. Conversely, $q_{-i} = 3$ can be followed by $q_{-i-1} = \bar{7}$, and 3.7 is closer to 2 than to 3: these two examples show that a possible wrong choice of q_{-i} can be compensated by q_{-i-1} . Indeed we hesitate between $q_{-i} = 2$ and $q_{-i} = 3$ when $\beta R_i / D$ is close to 2.5 (the middle between 2 and 3). As long as the exact quotient is between 2.3 and 2.7, either $q_{-i} = 2$ or $q_{-i} = 3$ will do.

Let us formalize this. In radix β , we define a signed redundant digit set as

$$\mathcal{D} = \{\bar{\gamma}, \dots, 0, \dots, \gamma\} \quad (9.27)$$

where γ is an integer such that $\gamma \geq \beta/2$ so that D contains at least $\beta + 1$ digits (for instance, in decimal, γ must be at least 5).

We now have to define a constraint on the partial remainder similar to (9.24), with the hope that it will yield a constraint on the next quotient digit similar to (9.25), but relaxed. Let us look for an iteration-invariant constraint of the form

$$-\rho D \leq R_i \leq \rho D \quad (9.28)$$

for some real $\rho > 0$. We need to ensure that we also have $-\rho D \leq R_{i+1} \leq \rho D$. Since $R_{i+1} = \beta R_i - q_{-i} D$, consider the maximum value that βR_i can take: it is $\beta \rho D$. In this case we will use the maximum possible value of q_{-i} , that is, $q_{-i} = \gamma$. The constraint $R_{i+1} \leq \rho D$ becomes in this case $\beta \rho D - \gamma D \leq \rho D$ or $\rho \leq \frac{\gamma}{\beta-1}$. Performing the symmetrical calculation on the other bound yields the same result. The larger ρ , the better in (9.28); therefore let us define ρ , the *redundancy factor* of a radix- β representation with digits from $-\gamma$ to γ :

$$\rho = \frac{\gamma}{\beta - 1}. \quad (9.29)$$

As $\gamma \geq \beta/2$, we have $\rho > 1/2$.

The interval of R_i defined by (9.28) is of size $2\rho D$: since $\rho > 1/2$, it is always larger than the size D that we had in (9.24), so this indeed is a relaxed constraint—the fact that it is symmetrical around zero is a natural consequence of the use of a symmetrical digit set.

Now, imposing (9.28) on $R_{i+1} = \beta R_i - q_{-i}D$ leads to the following constraint on the next quotient digit:

$$-\rho + \frac{\beta R_i}{D} \leq q_{-i} \leq \rho + \frac{\beta R_i}{D}. \quad (9.30)$$

This constraint replaces (9.25) and is also relaxed: a unique choice of q_{-i} such as $\left\lfloor \frac{\beta R_i}{D} \right\rfloor$ would correspond in (9.30) to $\rho = 1/2$, but with a redundant system, we have $\rho > 1/2$.

Therefore, when $\frac{\beta R_i}{D}$ is close to the middle between two integers, there are two possible choices of q_{-i} . What is more, the larger ρ (the more redundancy in the number system), the wider the interval around $\frac{\beta R_i}{D}$ where we have the freedom to choose among two values of q_{-i} .

In other terms, (9.30) shows that we may use an approximation of $\frac{\beta R_i}{D}$ to determine q_{-i} . Redundancy makes the quotient digit selection easier.

Example 1: $\beta = 4, \gamma = 2$

This example is of practical significance since it has a nice property: with $q_{-i} \in \{-2, -1, 0, 1, 2\}$, the main iteration $R_{i+1} = \beta R_i - q_{-i}D$ simply adds or subtracts D , possibly shifted left one position (this echoes Cauchy's observation that signed digits can simplify multiplications). Since $\rho = 2/3$, there is nevertheless a fair amount of redundancy in this system.

Figure 9.5 shows in this case the so-called P-D diagram, which illustrates the constraint (9.30) on the next quotient digit as a function of D and βR_i . It shows in which region of the $(D, \beta R_i)$ plane it is possible to choose each digit, and it also shows how these regions overlap. The colored part of this diagram also shows the convergence domain of the chosen digit-recurrence algorithm. This convergence domain is clipped on this example with vertical lines at $D = 1/2$ and $D = 1$, corresponding to the possible values of a floating-point significand normalised to $[1/2, 1]$.

Armed with this P-D diagram, we may now actually design a division algorithm that only considers the leading digits of βR_i and D to decide which digit q_{-i} should be selected. Figure 9.6 shows a zoom in Fig. 9.5, with a superimposition of the discretization grid corresponding to considering only 3 bits of D and 5 bits of βR_i . Here we assume a usual binary representation

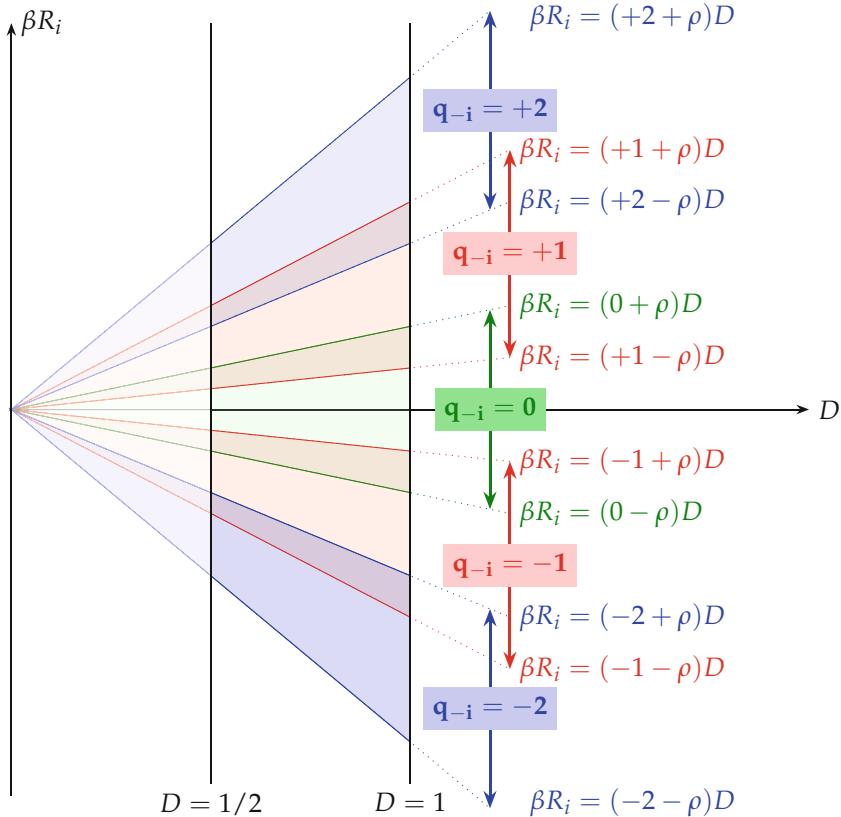


Fig. 9.5 P-D plot for radix 4 with $\gamma = 2$ hence $\rho = 2/3$, for the division of floating-point significands.

of D , and of course the bits input to the selection function do not include its constant leading 1 of weight 2^{-1} . For βR_i we assume a two's complement binary representation. Only R_0 , like D , is known to be a positive significand with a constant leading 1.

For any $(D, \beta R_i)$ point on the plane, truncating D to its 3 MSBs and truncating βR_i to its 5 MSBs will yield the point of the grid located in the bottom-left corner of the small rectangle enclosing $(D, \beta R_i)$. Or, a point of the grid is a coarse representative (on 3+5 bits) of all the possible values $(D, \beta R_i)$ of the rectangle immediately above right (excluding the upper and right borders of this rectangle). If the whole of such a rectangle is contained in the selection domain of one of the digits, then it is possible to select this digit as the next q_{-i} , based only on the value of these 3+5 bits. This will ensure the convergence of the algorithm.

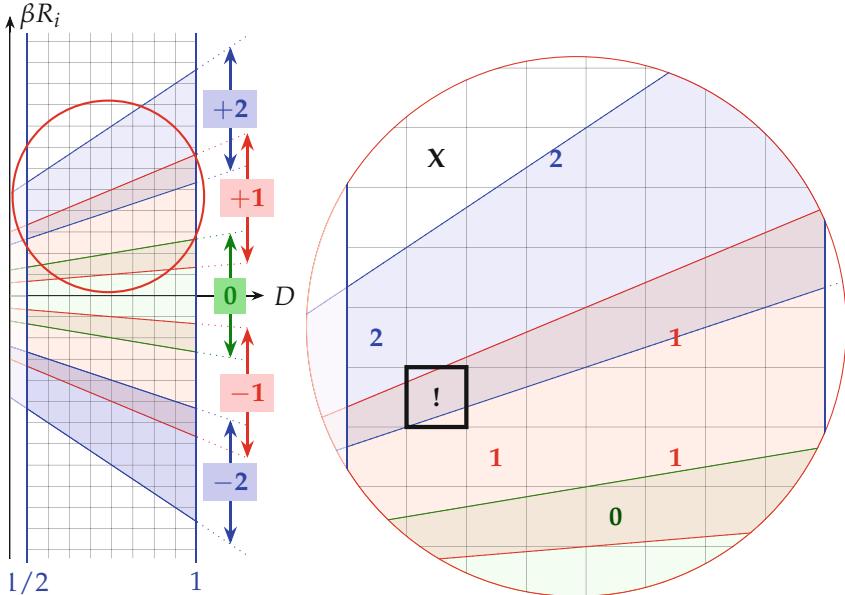


Fig. 9.6 Quotient digit selection by looking at 5 bits of βR_i and 3 bits of D .

In Fig. 9.6 we show several rectangles where quotient selection is possible. For instance, for the few rectangles labeled **1**, the rectangle is fully between the two red lines defining the selection interval for 1; these lines themselves are a representation of constraint (9.30). The fully white rectangles are outside of the convergence domain and can therefore be associated any value (we use an X as the usual “don’t care” symbol). However, any rectangle including at least one point in the convergence domain must be labeled with a proper digit³ (e.g., the topmost **2** on Fig. 9.6).

There remains in Fig. 9.6 one rectangle (labeled with !) whose top-left corner only belongs to the selection domain of 2 while its bottom-right domain only belongs to the selection domain of 1. It is impossible to label it with either digit, and therefore quotient digit selection cannot be decided on the basis of only 3 bits of D and 5 bits of βR_i .

Conversely, Fig. 9.7 illustrates a truncation of D and βR_i to, respectively, 3 and 6 bits. Here each small rectangle can be labeled with a digit. There are even several rectangles (labeled with **1|2**) where two digits are possible.

Figure 9.8 describes the corresponding architecture. Each radix-4 quotient digit q_{-i} is represented in two’s complement on three bits:

$$q_{-i} = -2^2 q_{-i,2} + 2^1 q_{-i,1} + 2^0 q_{-i,0} \quad (9.31)$$

³ The well-known Pentium FDIV bug, in 1995, was due to a handful of these border cells being improperly labeled.

which can be rewritten

$$q_{-i} = -4q_{-i}^{\ominus} + q_{-i}^{\oplus} \quad (9.32)$$

with $q_{-i}^{\ominus} = q_{-i,2} \in \{0, 1\}$ and $q_{-i}^{\oplus} = 2q_{-i,1} + q_{-i,0} \in \{0, 1, 2, 3\}$.

This architecture is also of practical significance on FPGAs, as the multiplexer along with the adder/subtractor can be merged in a single row of 6-input LUTs linked by fast carry logic (for more details see Sect. 5.4). The 6 inputs to each LUT are one bit of R_i , one bit of D , one bit of $2D$, and the three bits of q_{-i} .

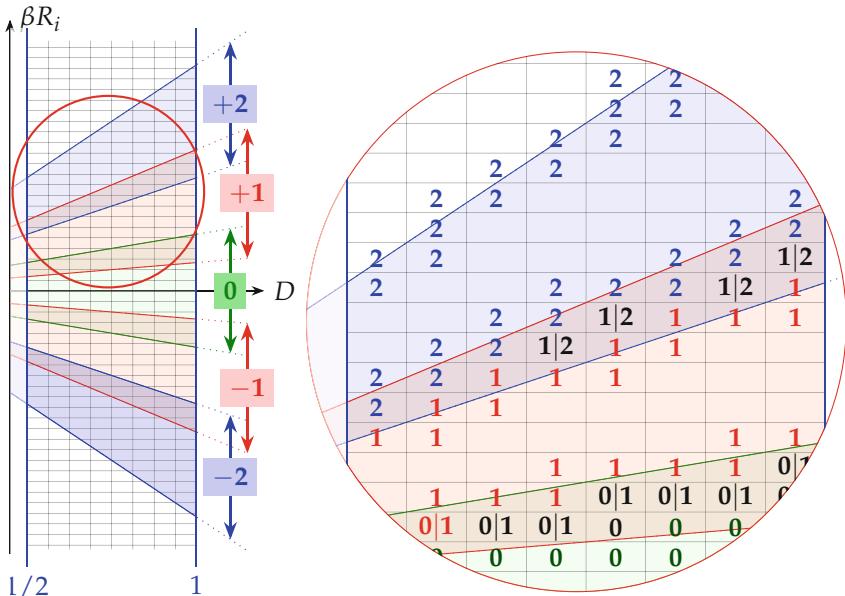


Fig. 9.7 Quotient digit selection by looking at 6 bits of βR_i and 3 bits of D .

Hands on: A divider with radix $\beta = 4$ and $\gamma = 2$ in FloPoCo

The following command produces a floating-point divider using the selection function in Fig. 9.7 and unrolling in space the architecture shown in Fig. 9.8.

```
flopoco FPDiv wE=8 wF=23 srt=42
```

Don't cares are currently not supported in FloPoCo; therefore all the values out of the convergence domain are assigned +2 or -2.

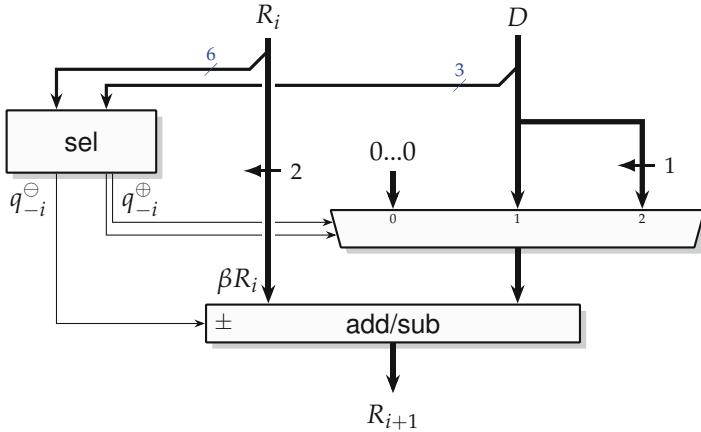


Fig. 9.8 Architecture of one step of a radix-4 divider with $\gamma = 2$. The constant shifts are represented by horizontal arrows with the shift value on the right (here 1 and 2 bits), to emphasize that they only consist of wiring. By construction the three leading bits of the addition result will be identical (sign bit). Therefore, two of them do not need to be computed, and R_{i+1} has the same size as R_i , also the size of all the thicker wires.

Example 2: $\beta = 4, \gamma = 3$

This is an example of a maximally redundant digit set. Figure 9.9 gives the P-D diagram, and Fig. 9.10 describes the corresponding architecture. As Fig. 9.9 shows, the digit selection intervals fully overlap in this case. More redundancy allows for a coarser grid for the digit selection approximation, hence, a much smaller selection function (it has 5 input bits and matches well, for instance, an FPGA LUT). However, the multiplexer now has one more input for 3D, which on FPGAs with 6-input LUTs prevents packing the multiplexer and the adder/subtractor in a single row of LUTs.

Note that the value of $3D = D + (D \ll 1)$ can be pre-computed only once, before the loop. In a fully pipelined (unrolled) FPGA implementation, we have the choice between (a) transmitting only D from step to step and recomputing $3D$ locally and (b) pre-computing $3D$ and transmitting D and $3D$ from step to step (the option illustrated in Fig. 9.10). Area-wise, it is essentially a trade-off between a row of registers and a row of LUTs to perform the addition. Delay-wise, however, the local recomputation adds to the critical path.

Hands on: A divider with radix $\beta = 4$ and $\gamma = 3$ in FloPoCo

The following command produces a floating-point divider using the selection function in Fig. 9.9 and the architecture in Fig. 9.10.

```
flopoco FPDiv wE=8 wF=23 srt=43
```

FloPoCo also includes a maximally redundant radix-8 variant ($\beta = 8, \gamma = 7$), but it is both larger and slower than the radix-4 variants. This is

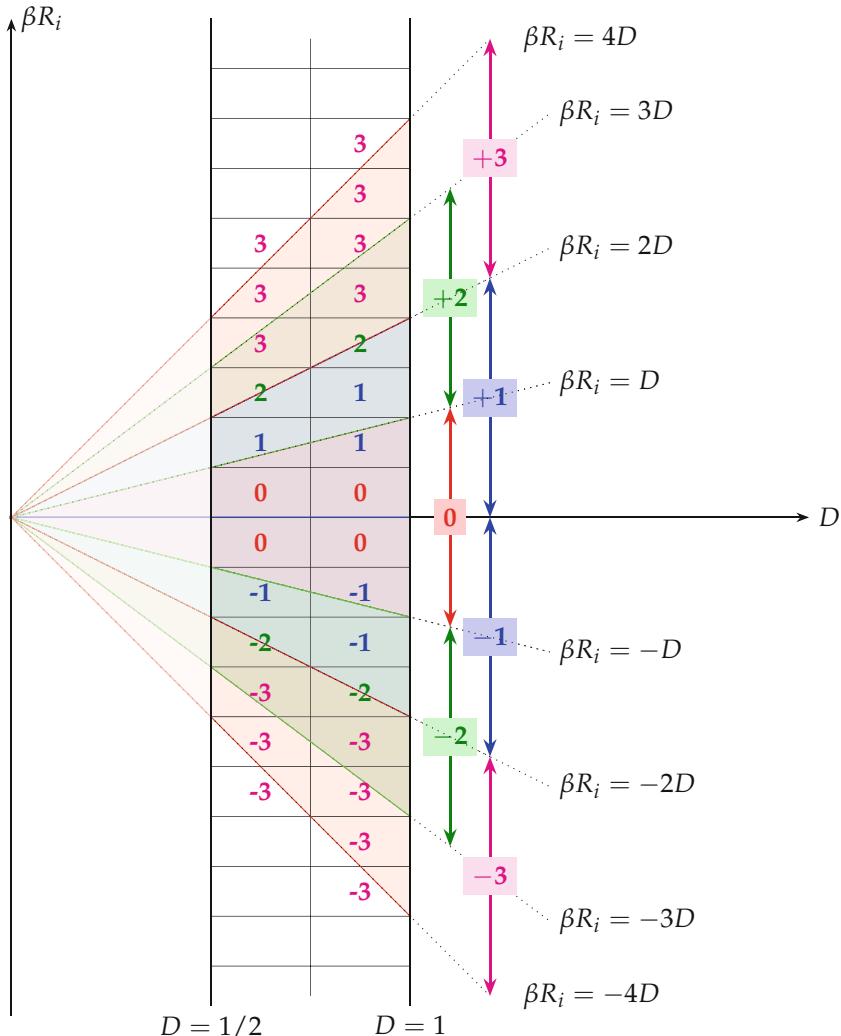


Fig. 9.9 P-D plot of a maximally redundant digit set, here radix 4 with $\gamma = 3$. In this case quotient digit selection is possible by considering 4 bits of βR_i and 1 bit of D .

consistent with an earlier extensive comparison of several SRT variants on FPGAs with 4-input LUTs [SBD04].

The previous two radix-4 detailed examples make sense for FPGAs for small mantissa sizes (up to single precision): there, the delay of a full carry-propagate add/sub is comparable to the delay of a LUT thanks to fast carry logic. In VLSI, or even on FPGAs for larger precisions, this will not be true,

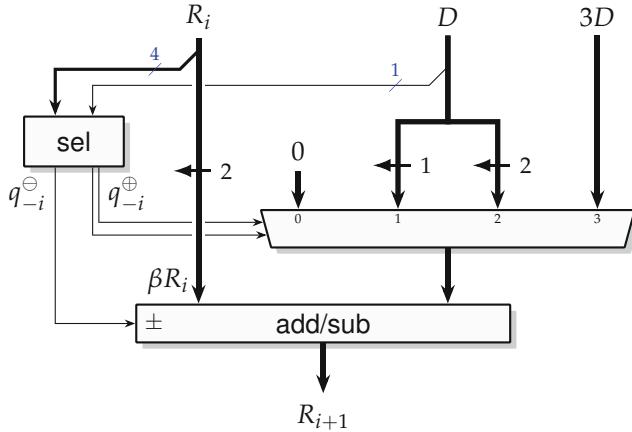


Fig. 9.10 Architecture of one step of a radix-4 divider with $\gamma = 3$. By construction the three leading bits of the addition result will be identical (sign bit). Therefore, two of them do not need to be computed, and R_{i+1} has the same size as R_i .

and the subtraction $\beta R_i - q_{-i} D$ has to be sped up. Techniques for this will be reviewed in Sect. 9.2.9.

9.2.5 Initialization and Termination with Redundant Digits

So far we have focused on the core of the iteration, which computes q_{-i} and then $R_{i+1} = \beta R_i - q_{-i} D$.

To initialize this iteration, we must ensure that the convergence condition (9.28) is true from the first iteration. In Sect. 9.2.3 we defined $R_1 = X/2$ so that $0 \leq R_1 < D$. Similarly, we must here initialize R_1 with X , scaled by some power of two so that $-\rho D \leq R_0 \leq \rho D$. For a maximally redundant system ($\gamma = \beta - 1$ hence $\rho = 1$), the initialization $R_1 = X/2$ is still valid. In the other cases, $1/2 < \rho \leq 1$, therefore $R_1 = X/4$ must be used. The quotient will be scaled back by the same factor.

For example, consider the division of two normalized floating-point significands, in radix $\beta = 4$ with the digit set parameter $\gamma = 2$. In this case we have $\rho = 2/3$: the choice $R_1 = X/4$ ensures that $0 \leq R_1 \leq \rho D$. This implies that we now compute a quotient such that $Q < 1/2$. Its first radix- β digit is still q_{-1} , but the selection function that produces this first digit can be simplified, all the more as R_1 is always positive.

Let us now address termination. The iteration should stop when we have enough quotient digits or equivalently when the remainder is small enough. Similar to Eqs. (9.17)–(9.22), p. 270 in the integer case, unrolling the significand division iteration (9.26) leads to

$$R_{i+1} = \beta^i \left(R_1 - D \underbrace{\sum_{j=1}^i \beta^{-j} q_{-j}}_{Q_i} \right) \quad (9.33)$$

which can be rewritten

$$R_1/D = Q_i + \beta^{-i} R_{i+1}/D \quad . \quad (9.34)$$

This shows that $Q_i = 0.q_{-1} \dots q_{-i}$ is an approximation to the significand quotient and defines the error of this approximation as $\beta^{-i} R_{i+1}/D$. Since $|R_{i+1}/D| \leq \rho$ by construction (see (9.28)), this error is such that

$$|R_1/D - Q_i| \leq \rho \beta^{-i} \quad . \quad (9.35)$$

The iteration may stop as soon as this error is smaller than some target error bound. In detail, for a floating-point division, this target error bound must take into account the initial down-scaling of X by a factor 2 or 4 (since the computed quotient will be correspondingly up-scaled), the possible normalization of the quotient, and the possible prescaling that will be studied in Sect. 9.2.8.

However, correct rounding will be easy since no information was lost in the transformation of (X, D) into (Q_i, R_i) : indeed, (9.34) is an equation which is implemented exactly, without any rounding or approximation.

9.2.6 Conversion of the Quotient from Redundant Form to Binary

The quotient is computed so far as $\sum_j \beta^j q_j$ in a redundant number system, and we now address its conversion back to usual binary.

The encoding of each q_j should be selected first and foremost to allow for an efficient implementation of the recurrence. This is the case of the two's complement q_j encoding for $\beta = 4$ (see Figs. 9.8 and 9.10).

It turns out that this encoding also enables a first generic technique to convert $\sum_j \beta^j q_j$ to usual binary, using a single final subtraction. With the q_j expressed in two's complement as $q_j = -\beta q_j^\ominus + q_j^\oplus$ (where q_j^\oplus is a $\log_2 \beta$ -bit digit and q_j^\ominus is a single bit), the final conversion of the quotient is achieved

by

$$Q_i = \sum_j \beta^j (-\beta q_j^\ominus + q_j^\oplus) \quad (9.36)$$

$$= -\beta \sum_j \beta^j q_j^\ominus + \sum_j \beta^j q_j^\oplus \quad (9.37)$$

This is a plain fixed-point binary subtraction, as $\sum_j \beta^j q_j^\oplus$ is a positive binary fixed-point number obtained by concatenating all the q_j^\oplus , and $\sum_j \beta^j q_j^\ominus$ is a positive binary fixed-point number obtained by concatenating all the $\log_2 \beta$ -bit strings obtained by left-padding q_i^\ominus with $\log_2 \beta - 1$ zeroes. The result of this subtraction is known to be positive if we were dividing two positive significands. All this works whatever the value of β as long as the q_j is expressed in two's complement.

This solution costs one final subtraction, which adds to the latency of the architecture. To avoid this it is possible to perform the conversion *on the fly*, as soon as each digit is produced. The idea is to define Q_i as the value of $\sum_j \beta^j q_j$ in binary and to iteratively add $\beta^j q_j$ to Q_i as soon as it is produced. However, these additions of $\beta^j q_j$ should be mere concatenations: if they involve a carry propagation, in particular if the last one may involve a carry propagation, then there is no latency benefit with respect to the previous solution using a unique final subtraction.

Let us detail this approach in the case of significand division (the integer case is similar with different indices). The recurrence is

$$Q_0 = 0 \quad (9.38)$$

$$Q_i = Q_{i-1} + \beta^{-i} q_{-i} \quad (9.39)$$

$$= Q_{i-1} + \beta^{-i} (q_{-i}^\oplus - \beta q_{-i}^\ominus) \quad (9.40)$$

If we had only the q_i^\oplus , concatenation would be enough. The need for a subtraction comes from the existence of the sign bits q_i^\ominus , to be subtracted with weight β^{-i+1} in (9.40). A solution is therefore to maintain, along with each quotient Q_i , a pre-decremented quotient $Q_i^\ominus = Q_i - \beta^{-i}$, so that Q_{i-1}^\ominus provides the result of the subtraction of β^{-i+1} when needed, i.e., when $q_{-i} < 0$.

The following recurrence implements this idea, while building Q_i and Q_i^\ominus using only concatenations (and we leave it to the reader to show by induction that it maintains the identities $Q_i = \sum_j \beta^j q_j$ and $Q_i^\ominus = Q_i - \beta^{-i}$):

$$\begin{aligned} \text{if } q_{-i} > 0 & \begin{cases} Q_i = Q_{i-1} + \beta^{-i} q_{-i}^{\oplus} \\ Q_i^{\ominus} = Q_{i-1} + \beta^{-i} (q_{-i}^{\oplus} - 1) \end{cases} \end{aligned} \quad (9.41)$$

$$\begin{aligned} \text{if } q_{-i} = 0 & \begin{cases} Q_i = Q_{i-1} \\ Q_i^{\ominus} = Q_{i-1}^{\ominus} + \beta^{-i} (\beta - 1) \end{cases} \end{aligned} \quad (9.42)$$

$$\begin{aligned} \text{if } q_{-i} < 0 & \begin{cases} Q_i = Q_{i-1}^{\ominus} + \beta^{-i} q_{-i}^{\oplus} \\ Q_i^{\ominus} = Q_{i-1}^{\ominus} + \beta^{-i} (q_{-i}^{\oplus} - 1) \end{cases} \end{aligned} \quad (9.43)$$

This on-the-fly conversion requires in each iteration two multiplexers of the size of Q_i and two registers in a pipelined design (for Q_i and Q_i^{\ominus}): it is fast but expensive in area.

9.2.7 High-Radix Integer Division

Looking back at Fig. 9.9, it is obvious that the tiling with quotient digits presented there cannot work for values of D that are close to zero. Therefore, this technique cannot be used directly to divide two integers (it does not handle division by 1). There are at least two families of solutions:

- It is possible to perform quotient digit selection differently, by comparing in parallel βR_i with pre-computed multiples of D that each lies in the digit overlap region. A classic example is given in Fig. 9.11 for radix-4 with $\gamma = 2$. The multiples $\pm D/2$ and $\pm 3D/2$ are chosen because they are easy to compute. A problem with this solution is that its critical path now involves two carry propagations in sequence (a comparison is essentially a subtraction where we are only interested in the sign of the result: compared to a subtraction, it is possible to save some of the hardware that computes the subtraction result, but not the carry propagation). Therefore, it is unclear if there is a context where this approach is more efficient than the restoring division algorithm in Fig. 9.4, whose cost is also two carry propagations for two quotient bits. A better use of this technique could be with a non-redundant digit set. For instance, in radix-8, compute in parallel $8R_i - q_i D$ for $q_i \in \{0 \dots 7\}$, and select the first non-negative one. Delay-wise, one radix-8 iteration now involves only one carry propagation (using merged arithmetic for each $8R_i - q_i D$) plus a little logic (a priority encoder and an 8:1 multiplexer). Area-wise, however, this is a rather expensive solution.
- Another solution is to multiply both X and D by the same power of two 2^k , chosen such that the MSB of D is a 1 (since $D \neq 0$, this is always possible). This involves a leading zero counter and two shifters. This is very similar to computing a normalized floating-point representation of D , with k being the exponent.

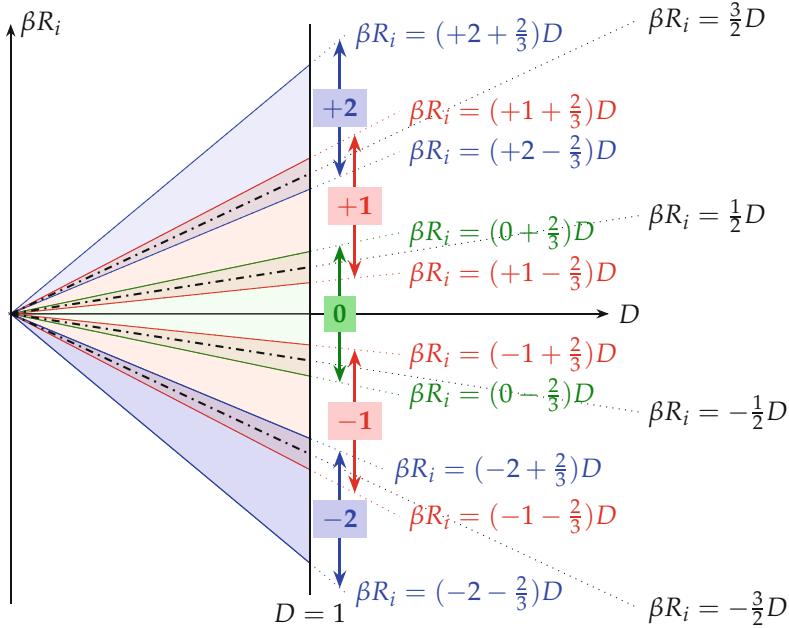


Fig. 9.11 Digit selection by comparison with divisor multiples ($\beta = 4, \gamma = 2$).

9.2.8 Prescaling

Since quotient digit selection is easier for larger values of the divisor D , another classic technique, called *prescaling*, consists of transforming the problem into another one where D is closer to 1. For this purpose, both X and D are multiplied by the same value. This, of course, does not change the quotient.

A first simple example is to multiply both X and D by $\frac{3}{2}$ when $X < \frac{5}{8}$. The factor $\frac{3}{2}$ is chosen because it is cheap to compute (one addition). The limit value $\frac{5}{8}$ is chosen so that it can be implemented as a 4-bit comparison (in an FPGA it would consume only one LUT) while ensuring the prescaled X never exceeds 1 since $\frac{3}{2} \times \frac{5}{8} = \frac{15}{16} < 1$. Figure 9.12a shows that this prescaling avoids the problematic case of Fig. 9.6: it is now possible to have a selection function that inputs only 5 bits of βR_i and 3 bits of D . Besides, it now has many more *don't care* values (for all the values of D smaller than $5/8$). In short, we have divided at least by two the area of the `sel` table in Fig. 9.8, used in each iteration, at the cost of two more parallel additions (one for D , one for X) before the iteration themselves. This trade-off may be interesting, or not.

Table 9.2 gives the parameter for a more powerful classic prescaling approach [EL90]. It determines, out of the 3 first non-trivial bits of D , an

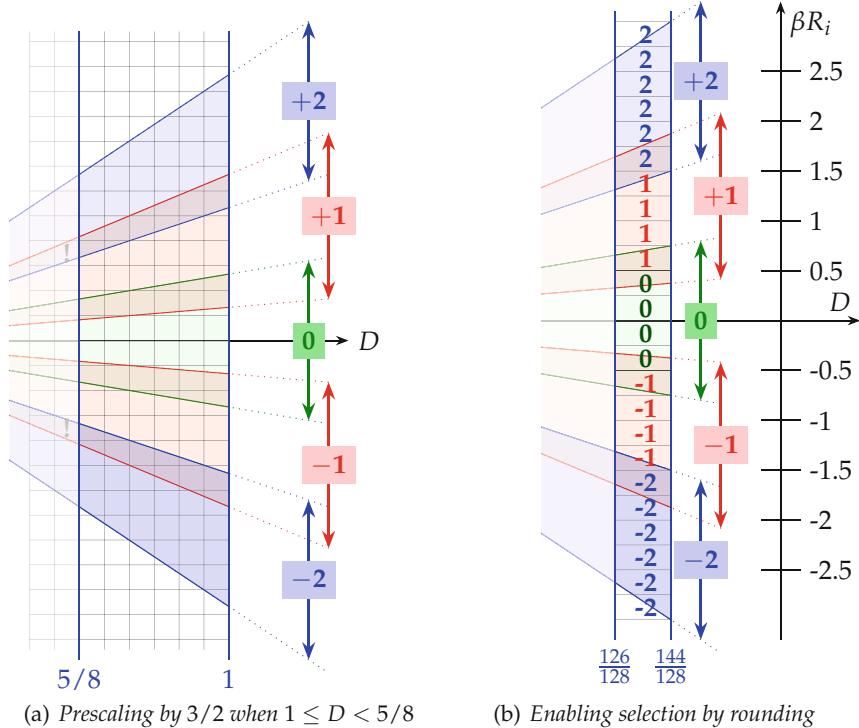


Fig. 9.12 Two examples of prescaling for radix 4 with $\gamma = 2$.

approximation to the inverse of $1/D$ that can be computed in two additions only, using the architecture in Fig. 9.13. After prescaling, the new divisor D_{scaled} always belongs to the interval $[\frac{126}{128}, \frac{144}{128}] = [\frac{63}{64}, \frac{9}{8}] = [0.984375, 1.125]$. In this range, the quotient digit selection function does no longer depend on D , as Fig. 9.12b shows. For this reason, it is not a problem that D_{scaled} can exceed 1 here: it will not add one bit to the (no longer existing) D input of the selection table.

Another point of view is that D_{scaled} is sufficiently close to 1 that we may assume that we divide by 1 (only for quotient digit selection purposes! The computation of the next partial remainder must still use the actual value of D_{scaled}).

A practical consequence is that with such prescaling, the next quotient digit can also be computed by a simple rounding of βR_i to the nearest digit (round to nearest with ties up is OK here). The intuition is that we are now almost dividing by 1. Again, see Fig. 9.12b. This will cost a 4-bit addition.

For such a small radix, this addition is not dramatically cheaper than a 5-input quotient digit selection table. However, since strong prescaling dispenses of a quotient digit selection table, it also enables very high radices

Table 9.2 Prescaling from [EL90].

D	$16D$	Scaling	Implementation	$128D_{\text{scaled}}$
.1000...	[8, 9)	2	$1 + 1/2 + 1/2$	[128, 144)
.1001...	[9, 10)	7/4	$1 + 1/2 + 1/4$	[126, 140)
.1010...	[10, 11)	13/8	$1 + 1/8 + 1/2$	[130, 143)
.1011...	[11, 12)	12/8	$1 + 1/2$	[132, 144)
.1100...	[12, 13)	11/8	$1 + 1/8 + 1/4$	[132, 143)
.1100...	[13, 14)	10/8	$1 + 1/4$	[130, 140)
.1100...	[14, 15)	9/8	$1 + 1/8$	[126, 135)
.1100...	[15, 16)	9/8	$1 + 1/8$	[135, 144)

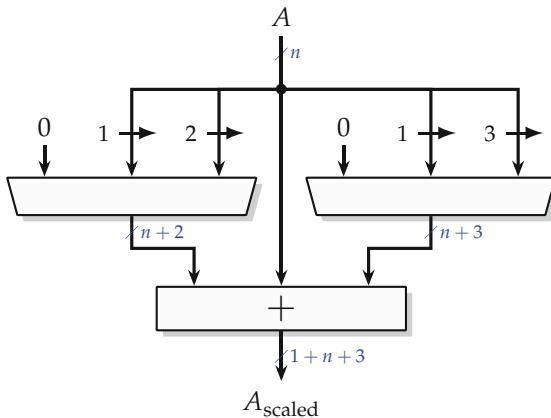


Fig. 9.13 Architecture prescaling by the factors of Table 9.2 to enable Fig. 9.12b. This architecture must be used both for $A = X$ and $A = D$. The multiplexer control signals implement Table 9.2 [EL90].

[ELM94; LCC06]. However, the computation of the product $q_i D$ in such an architecture requires an actual multiplier, albeit a rectangular one.

Prescaling simplifies the quotient selection function but entails some overhead.

- The main overhead is the prescaling hardware described in Fig. 9.13 and usually duplicated (for X and for D).
- There is another less obvious overhead due to the fact that both prescaled X and prescaled D must be computed exactly and then used exactly in the division iteration: this enlarges the iteration datapaths. For instance, Fig. 9.13 outputs a scaled number that requires one more bit on the MSB (as the maximum scaling factor is 2) and three more bits at the LSB (since all the bits of $A \gg 3$ should be kept).

- Finally, we still have to satisfy $|R_1| \leq \rho D$ at the beginning of the iteration. The initial scaling between X and R_1 (discussed in Sect. 9.2.5) must be adapted to the worst-case prescaling of X . For instance, if X was a floating-point significand such that $1 \leq X < 2$, after prescaling using the prescaling of Table 9.2, we have $\frac{126}{128} \leq D_{\text{scaled}} < \frac{144}{128}$ but also $\frac{9}{8} \leq X_{\text{scaled}} < 4$; therefore, for $\rho = 2/3$ we have to define $R_1 = 2^{-3}X$ to ensure $0 \leq R_1 \leq \rho D$. This may increase the number of iterations.

9.2.9 Speeding Up the Partial Remainder Computation

Last but not least, we discuss techniques that will speed up the (large) subtraction in $R_{i+1} = \beta R_i - q_{-i}D$. The main idea here is to keep R_i in carry-save format (see Sect. 5.2.6 and Chap. 7). Then, since q_{-i} fits on very few bits, $q_{-i}D$ can always be expressed as few additions, and the bit heap for $\beta R_i - q_{-i}D$ is not high: it can be compressed in very few compressor levels to a carry-save number (a bit-heap of height 2). This makes the computation of $\beta R_i - q_{-i}D$ very fast. Only the final R_i needs to be converted from carry-save to standard binary, to help decide rounding or to compute the final positive remainder.

However, keeping R_i in carry-save format negatively impacts quotient digit selection.

First, R_i is represented on twice as many bits in carry-save as in binary. This is a problem as we input bits of R_i to the quotient selection table: the table size becomes much larger. For instance, if R_i is in carry-save, the table in Fig. 9.8 has $6 + 6 + 3 = 15$ input bits instead of 9. A simple solution is to have a small, fast carry-propagate adder performing the conversion of the 6 leading bits of R_i into standard binary. However, these 6 bits won't be the same as those we would have if R_i was all kept in binary, because of the second problem: carry-save is a redundant format. Without performing a full-length conversion of R_i , we don't know what the carry-in of our small addition should be.

Fortunately, this uncertainty is easily modeled and injected in the P-D plots, and redundancy can again be exploited to design a selection function taking it into account. Let t be the bit position at which we truncate the carry-save value of βR_i before converting it into standard binary. The possible carry that we ignore when performing this addition would have the weight 2^t : the approximate value $\tilde{\beta R}_i$ that we will be using in the selection function is

$$\tilde{\beta R}_i = \beta R_i + \delta \quad \text{with } \delta = 0 \text{ or } \delta = -2^t. \quad (9.44)$$

This is a one-sided approximation: $\beta\tilde{R}_i \leq \beta R_i$. From (9.44) we also have $\beta\tilde{R}_i + 2^t \geq \beta R_i$. Therefore, to ensure constraint (9.30) which defined the quotient digit selection boundaries, we have to ensure the new constraint

$$-\rho + \frac{2^t}{D} + \frac{\beta\tilde{R}_i}{D} \leq q_{-i} \leq \rho + \frac{\beta\tilde{R}_i}{D}. \quad (9.45)$$

In other words the missing information from the ignored carry shrinks the selection interval. Figure 9.14 is an adaptation of Fig. 9.7 that illustrates this, using a truncation of βR_i at position $t = -4$. As this figure shows, there is still a large amount of overlap, but it is no longer possible to input only 6 bits of $\beta\tilde{R}_i$ to the selection function.

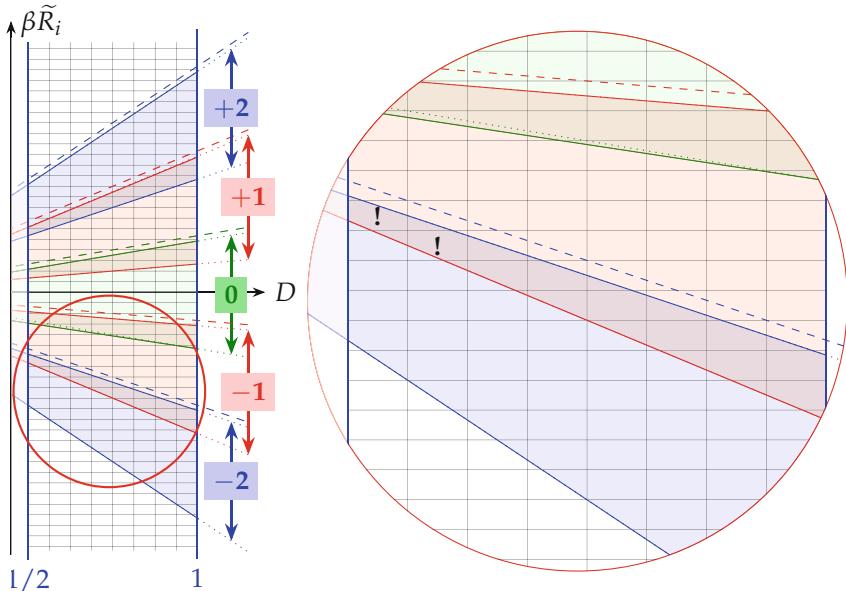


Fig. 9.14 Quotient digit selection using a truncated approximation $\beta\tilde{R}_i$ of the carry-save βR_i . The dashed lines show the case when βR_i is used exactly. Looking at 6 bits of $\beta\tilde{R}_i$ and 3 bits of D is no longer enough.

On ASIC, as noted by Parhami [Par10], there is far more speedup potential in computing the partial remainder in a digit-parallel way than there is in reducing the iteration count by going from radix-2 to a higher radix. This is therefore the main motivation of a redundant system for the quotient, and a radix-2 redundant system makes perfect sense. Indeed, an old survey paper [HOH97] comparing several implementations of radix-2 and radix-4 dividers concluded that *divider performance is only weakly sensitive to*

reasonable choices of architecture but significantly improved by aggressive circuit techniques.

On FPGAs, fast carry logic (see Sect. 5.4, p. 125) with the digit product merged in the addition LUTs makes the computation of the partial remainder very area-efficient (see again Fig. 9.8) and fast enough for sizes up to 32 bits. For larger precisions, a high-frequency computation of the partial remainder may be achieved with little overhead by keeping it in high-radix carry-save (HRCS) (see Sect. 5.2.6, p. 110). The main overhead will not be in the extra registers needed (1/16 for a HRCS chunk size of 16) but in the larger quotient digit selection function (Fig. 9.14).

9.2.10 A Case Study for Low-Latency Double Precision Division

To conclude this section, the interested reader will find in [Bru18] a detailed description of a state-of-the-art divider combining most of the above techniques. It is presented as a radix-64 divider, but each cycle actually consists of three radix-4 steps ($64 = 2^{2 \times 3}$). It keeps R_i in carry-save form. Thus, after the prescaling of Table 9.2, the input to the selection function consists of 6 bits of $\beta\tilde{R}_i$. The latency is further reduced by aggressive (and area-expensive) speculation techniques, for instance, all the possible values of $\beta R_i - q_{-i}D$ are computed in parallel while q_{-i} is being determined; then a multiplexer controlled by q_{-i} selects the correct one.

9.3 Division Using Multiplication by the Reciprocal

We now present a completely different family of division techniques that multiply by X an approximation to the reciprocal $1/D$, itself possibly obtained using multiplicative algorithms.

Fixed-point reciprocal approximation techniques cannot work well if D is allowed to be close to 0. It is not a problem in a floating-point division (detailed in Sect. 11.3, p. 347) where a fixed-point divider inputs two floating-point significands: in this case, $D \in [1, 2)$. In the sequel, we develop this use case and therefore assume that X is also a significand: $X \in [1, 2)$.

Floating-point division is indeed the main application of reciprocal-based techniques [OF97b; SL97; Obe99; Mar00; PB02; WBL06; GES07; Pas12; Jai+14]. An ad hoc fixed-point divider can also be built out of the techniques presented in this section, but it will essentially consist in wrapping the architectures presented here in fix-to-float and float-to-fix converters, the latter being built out of shifters and leading-zero counters surveyed in Chap. 10.

The management of the floating point itself (exponent processing, normalization, etc.) is detailed in Sect. 11.3, p. 347, which also defines an accuracy goal for the fixed-point divider in a faithful floating-point divider:

$$|\delta_Q| = |Q - X/D| < 2^{-w_F-2} \quad . \quad (9.46)$$

Here, the w_F parameter defines the fraction size in a floating-point format (see Sect. 2.5, p. 48).

9.3.1 Error Analysis for a Faithfully Rounded Quotient out of a Reciprocal Approximation

Figure 9.15 shows a generic architecture of a divider using multiplication by a reciprocal approximation R .

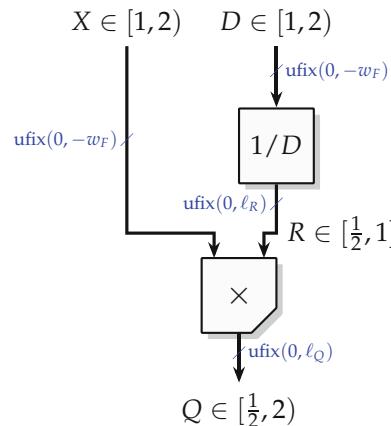


Fig. 9.15 Architecture of a divider using an approximation to the reciprocal.

The fixed-point formats of the inputs are known. The MSB parameters of the formats of R and Q on the figure are deduced from their respective intervals. For R , we need the bit of weight 0 only for the reciprocal of 1, which seems wasteful: in the sequel, we will investigate ways to save this bit. For Q , our astute reader should question the $\text{ufix}(0, \ell_Q)$ format: since Q is an approximation to X/D , isn't it possible that the approximation error added to the exact quotient entails that the value of Q reaches $Q = 2$, which would require one more bit at the MSB? The answer to this question is no. In our divider, the largest quotient is achieved when dividing $2 - 2^{-w_F}$ (largest possible significand) by 1, in which case the exact result is $2 - 2^{-w_F}$. Adding an error smaller than 2^{-w_F-2} , it is impossible to reach 2.

Let us now determine the value of the parameters ℓ_R and ℓ_Q of this figure (using a variation of the techniques developed in Sect. 8.1.3). The absolute error δ_Q can be defined as

$$\delta_Q = Q - \frac{X}{D} \quad (9.47)$$

$$= Q - XR + XR - \frac{X}{D} \quad (9.48)$$

$$= \underbrace{Q - XR}_{=\delta_{\text{mult}}} + \underbrace{X \left(R - \frac{1}{D} \right)}_{=\delta_{\text{recip}}}. \quad (9.49)$$

Therefore,

$$|\delta_Q| \leq |\delta_{\text{mult}}| + |X| \times |\delta_{\text{recip}}| \quad (9.50)$$

and since $|X| < 2$,

$$|\delta_Q| < |\delta_{\text{mult}}| + 2|\delta_{\text{recip}}| \quad (9.51)$$

If we assume a faithful multiplier and a faithful reciprocal approximation, then $|\delta_{\text{mult}}| < 2^{\ell_Q}$ and $|\delta_{\text{recip}}| < 2^{\ell_R}$. Choosing a faithful multiplier to $\ell_Q = -w_F - 3$ and a faithful reciprocal approximation to $\ell_R = -w_F - 4$ ensures the bound (9.46). If the reciprocal is tabulated, then it can be correctly rounded to the nearest for $\ell_R = -w_F - 3$ for the same accuracy.

Conversely, when $\ell_R = -w_F - 4$, it is sometimes possible to save the MSB bit of R , so that R is actually in the ufix($-1, -w_F - 4$) format. For this, it is enough that the value returned for the input 1 is $1 - 2^{-w_F - 4}$. Then the error of the $\boxed{1/D}$ operator reaches $2^{-w_F - 4}$ on this input: strictly speaking, this block is no longer faithful. However, the error of the full fixed-point divider remains strictly smaller than $2^{-w_F - 2}$.

The architecture presented so far outputs a result accurate to $2^{-w_F - 2}$ in an output format of precision $\ell_Q = -w_F - 3$. An alternative would be a faithful divider to a format of precision $2^{-w_F - 2}$. It is easy to build, using the generic architecture in Fig. 3.4, p. 83. The final round step will entail an error bounded by $2^{-w_F - 3}$, so the error budget for the architecture in Fig. 9.15 becomes $\delta_Q = 2^{-w_F - 3}$ instead of $2^{-w_F - 2}$. With the same error analysis, we must add one bit to ℓ_Q and one bit to ℓ_R . The final rounding itself is for free (the rounding bit can be merged in the compressor tree of the multiplier), but this option adds one bit to R , a multiplier input: this probably costs more than adding one bit to the output which, in a floating-point divider, is input to a multiplexer.

The construction of the truncated multiplier in Fig. 9.15 was addressed in Chap. 8. The remainder of this section addresses the construction of the $\boxed{1/D}$ block, first using generic methods and then using methods that are specific to the reciprocal.

9.3.2 Reciprocal Using a Generic Approximation Method

Part III of this book deals with generic function approximation techniques which can be used with the function $f(x) = 1/x$ on the interval $[1, 2)$. We refer the interested reader to this part for details. In general, these techniques should be considered for low precision.

The relevance of these techniques is highly dependent on the capabilities of the target in terms of multipliers, but also of embedded tables. Pasca [Pas12] showed that for single precision on DSP-enabled FPGAs, a piecewise polynomial approximator of degree 2 for the reciprocal was the most attractive option. For double precision, however, he found that evaluating $1/D$ using techniques specific to the reciprocal was more efficient than using generic techniques. Let us now review these techniques.

9.3.3 Reciprocal Using Newton-Raphson Iteration

In general, the Newton-Raphson method finds the zero of some function $f(x)$ using the following iteration:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad (9.52)$$

with suitable initial guess x_0 . To compute the reciprocal of D , the function whose zero we need to find is $f(r) = 1/r - D$. This leads to the following recurrence:

$$r_{i+1} = 2r_i - D \times r_i^2. \quad (9.53)$$

There is one square and one multiplication in this iteration, and they must be computed in sequence.

Let us define the reciprocal remainder⁴ as

$$e_i = 1 - Dr_i. \quad (9.54)$$

Computing e_{i+1} by rearranging (9.54) to r_i and plugging the result in (9.53), yielding

$$e_{i+1} = 1 - Dr_{i+1} = e_i^2. \quad (9.55)$$

From (9.55), we conclude that the error bound is squared or in other terms that the number of significant bits doubles at each iteration as long as the

⁴ Note that it is also the opposite of the relative error of r_i with respect to $1/D$, as $\varepsilon_i = \frac{r_i - \frac{1}{D}}{\frac{1}{D}} = Dr_i - 1 = -e_i$.

initial error e_0 is less than one (quadratic convergence). The same holds for the absolute error, since $\delta_i = -\frac{1}{D}e_i$ and $D \in [1, 2]$.

A consequence of (9.55) is that the iteration will converge as soon as $e_0 < 1$. In general, an architecture may use an initial approximation $r_0 \approx 1/D$ read from a table inputting the leading bits of D . Multiplierless table-based methods (studied in Chap. 17) can also be used – they were invented for reciprocal approximations [DM95] among other [Sun+84]. For double precision division, Pasca [Pas12] used a degree-2 piecewise polynomial to provide the initial approximation to a Newton-Raphson step.

Another consequence is that $r_i \leq 1/D$, since $e_i \geq 0$. This is welcome as it will help save the bit of weight zero of R in Fig. 9.15.

All the previous assumed an exact computation of (9.55), i.e., without any rounding errors. Let us now address implementation issues in such a way that the nice properties of the exact computation are preserved. Figure 9.16b shows the corresponding architecture. We use upper-case letters for the fixed-point values computed by the architecture: R_i is the approximation to r_i computed by the architecture, with $R_0 = r_0$ since this value can be known perfectly by exhaustive enumeration.

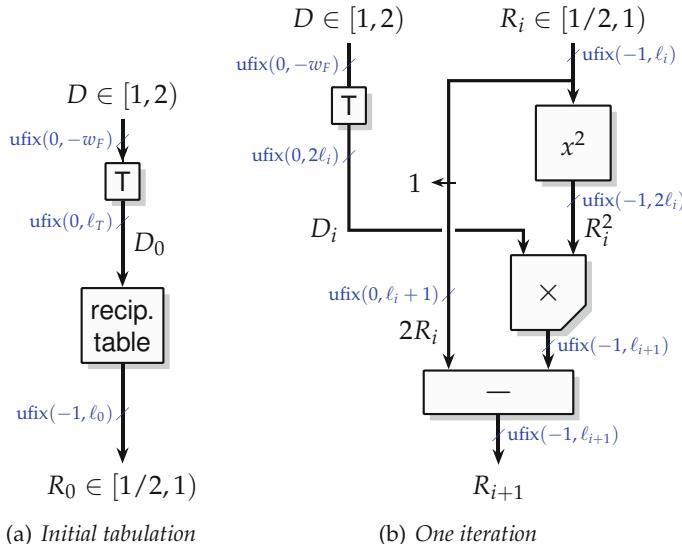


Fig. 9.16 Newton-Raphson reciprocal iteration.

A first implementation issue is indeed the initial approximation R_0 . It is desirable that $R_0 \leq 1/D$, but also that R_0 never reaches 1 (to save the bit of weight 0). Let us assume R_0 is read from a table, as illustrated in Fig. 9.16a (again there are other options, such as the multiplierless table-based methods studied in Chap. 17). Then, it is also desirable to have as few bits of D as

possible input to the table. The cheapest option is to use a truncation of the lower bits of D :

$$D_0 = \lfloor D \rfloor_{\ell_T} \in \text{ufix}(0, \ell_T) \quad (9.56)$$

for some parameter ℓ_T to be fixed later. Then we have $1 \leq D_0 \leq D \leq 2 - 2^{\ell_T}$. The problem is then that $1/D_0 \geq 1/D$. To fulfill the goal $R_0 \leq 1/D$, the table should therefore store

$$R_0 = \left\lfloor \frac{1}{D_0 + 2^{\ell_T}} \right\rfloor_{\ell_0} . \quad (9.57)$$

This formulation ensures $R_0 \leq 1/D$ and $1/2 \leq R_0 < 1$. Due to the two truncations involved, R_0 is not a faithful approximation of $1/D$. Indeed, the initial reciprocal remainder can be rewritten as

$$e_0 = 1 - DR_0 \quad (9.58)$$

$$= 1 - (D_0 + 2^{\ell_T})R_0 + (D_0 + 2^{\ell_T})R_0 - DR_0 \quad (9.59)$$

$$= (D_0 + 2^{\ell_T}) \left(\frac{1}{D_0 + 2^{\ell_T}} - R_0 \right) + (D_0 + 2^{\ell_T} - D)R_0. \quad (9.60)$$

Now, each of the terms can be bounded as follows:

$$e_0 = \underbrace{(D_0 + 2^{\ell_T})}_{\leq 2} \underbrace{\left(\frac{1}{D_0 + 2^{\ell_T}} - R_0 \right)}_{< 2^{l_0}} + \underbrace{(D_0 - D + 2^{\ell_T})}_{< 0} \underbrace{R_0}_{< 1}. \quad (9.61)$$

The $D_0 + 2^{\ell_T} \leq 2$ and $D_0 - D < 0$ follow from (9.56), while $\frac{1}{D_0 + 2^{\ell_T}} - R_0 < 2^{l_0}$ follows from (9.57). Ignoring the correlation between D_0 and R_0 , we get

$$0 \leq e_0 < 2 \cdot 2^{\ell_0} + 2^{\ell_T} . \quad (9.62)$$

A sensible choice is therefore $\ell_T = \ell_0 + 1$, which gives $0 \leq e_0 < 4 \cdot 2^{\ell_0}$. Now, we may also consider the correlation between D_0 and $R_0 \approx 1/D_0$: they cannot both be maximal together in (9.60). The actual bound for e_0 is actually closer to $3 \cdot 2^{\ell_0}$ when $\ell_T = \ell_0 + 1$ (we leave this as an exercise for the reader).

Back to the Newton-Raphson iteration, let us now consider the implementation of the iteration itself (9.53). As shown in Fig. 9.16b, it is possible to use a squarer, which is cheaper than a multiplier (see Chap. 14), and of course to implement the multiplication by two as a shift.

The quadratic convergence (9.55) assumes an exact computation (without rounding errors). To conserve near-quadratic convergence, an architecture should be designed such that the rounding errors are themselves of the or-

der of e_i^2 . The detailed error analysis is left as an exercise to the reader (a sketch is given in Sect. 9.4 for a slightly different algorithm), but it is easy to provide a few guidelines by observing Fig. 9.16b.

- Since the error grows quadratically, an architecture computing just right should aim for $\ell_{i+1} \approx 2\ell_i$ and $\delta_i \approx 2^{\ell_i}$.
- ℓ_0 will be defined by the tabulated initial approximation.
- There is no point in using a truncated squarer, since it would prevent R_{i+1} to be twice as accurate as R_i . Therefore, in Fig. 9.16b, the output of the squarer is the exact R_i^2 , and its size is twice that of R_i .
- It is useful to truncate D in the early iterations, as it does not need to be more accurate than R_i^2 . Conversely, in the last iteration, the truncation box \boxed{T} does nothing, and D is input directly to the multiplier.
- A truncated multiplier must be used, as an exact one would entail a much larger output with little accuracy gain.

Altogether, we have only two rounding errors in Fig. 9.16b: the truncation of D , if any, and the truncation of the multiplier result.

One option for the $\boxed{x^2}$ box is to tabulate it (see Chap. 17). This option is attractive for the first iteration, all the more as it opens an optimization opportunity. If we tabulate the square, we then have two consecutive tables, the one that produces R_0 and the one that inputs R_0 to produce R_0^2 . Instead of these two tables in sequence, it is more efficient, for the same cost, to replace the second table with a table that inputs D_0 and directly outputs an approximation of R_0^2 . This saves the latency of one table.

9.4 Division Using Quadratic Series Expansion

Another multiplicative recurrence can be derived from the geometric series

$$\sum_{n=0}^{\infty} x^n = \frac{1}{1-x} \text{ for } |x| < 1 \quad (9.63)$$

as follows [CHT02]. Let $R_0 \approx 1/D$ (again, read from a table), and let $e_0 = 1 - R_0 D$ the relative error of this initial approximation. Then

$$\frac{1}{D} = \frac{R_0}{1 - e_0} \quad (9.64)$$

$$= R_0(1 + e_0 + e_0^2 + e_0^3 + \dots) \quad (9.65)$$

$$= R_0(1 + e_0)(1 + e_0^2)(1 + e_0^4)(1 + e_0^8) \dots \quad (9.66)$$

The method we present now is really based on the rewriting of (9.65) as (9.66). Each new factor in (9.66) doubles the accuracy of the product

(quadratic convergence again). Therefore, (9.66) can be implemented as a recurrence that computes $e_i = e_0^{2^i}$ out of e_0 and uses it to compute r_i :

$$r_0 = R_0 \approx \frac{1}{D} \quad (9.67)$$

$$e_0 = 1 - r_0 D \quad (9.68)$$

$$r_{i+1} = r_i \times (1 + e_i) \quad \text{for } n \in 0, \dots, n-1 \quad (9.69)$$

$$e_{i+1} = e_i^2 \quad \text{for } n \in 0, \dots, n-2 \quad (9.70)$$

Furthermore, a slight modification directly computes the quotient:

$$r_0 = R_0 \approx \frac{1}{D} \quad (9.71)$$

$$e_0 = 1 - r_0 D \quad (9.72)$$

$$q_0 = Xr_0 \quad (9.73)$$

$$q_{i+1} = q_i \times (1 + e_i) \quad \text{for } n \in 0, \dots, n-1 \quad (9.74)$$

$$e_{i+1} = e_i^2 \quad \text{for } n \in 0, \dots, n-2 \quad (9.75)$$

In the following, we discuss further the complete quotient formulation (9.71)–(9.75), but this whole discussion applies similarly to the reciprocal computation (9.67)–(9.70). An architecture for (9.71)–(9.75) is given in Fig. 9.17. Again, we use upper-case letters Q_i and E_i for the fixed-point values approximating q_i and e_i by the architecture. Figure 9.17 also includes three truncation blocks $\overline{\square}$. They will be justified by the error analysis below, which will also help determine all the parameters on this figure. These truncation steps just drop bits and are thus for free. They are also optional if the mapping to the target hardware allows, in particular on FPGAs, when the fixed-size multipliers and RAM blocks have sufficiently large input.

From an architectural point of view, since R_0 fits on a few bits, multiplying X by R_0 is cheaper than a later multiplication by the final reciprocal result R_i . Besides, this computation of Q_0 can occur in parallel with the computation of E_0 , so its latency is completely hidden, as illustrated in Fig. 9.17a.

As Fig. 9.17b shows, E_{i+1} can be computed, while Q_{i+1} is being computed. One step therefore has a shorter latency than a Newton-Raphson step, whose latency is that of a squarer followed by a multiplier (see Fig. 9.16b). Considering that the multiplication by X also entails a shorter latency, this solution offers a much shorter critical path for the complete division than Newton-Raphson.

Let us now sketch the fixed-point formats of the various data involved in an efficient⁵ implementation.

⁵ In Fig. 9.17a, we have an architecture where both X and D are multiplied by the same R_0 that is an approximation to the inverse of D . This is exactly what we called *prescaling*

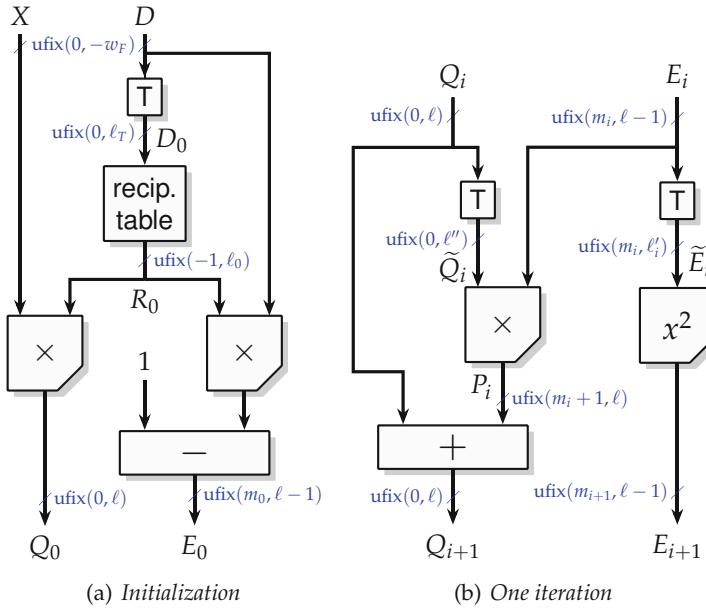


Fig. 9.17 Division by quadratic series expansion.

- By definition, $R_0 D$ is close to 1; therefore, E_0 is close to 0. For instance, in the previous section, it was bounded as $E_0 < 2^{\ell_0+2}$ where ℓ_0 is the LSB of the initial reciprocal approximation R_0 . More generally, if $E_0 < 2^{m_0}$, then ignoring the rounding errors, we would have $E_1 < 2^{2m_0}$, $E_2 < 2^{4m_0}$, and $E_i < 2^{2^im_0}$. This defines the MSB of E_i as

$$m_i = 2^i m_0 \quad . \quad (9.76)$$

This still holds in the presence of rounding errors if the E_i datapath (on the right of Fig. 9.17b) uses a truncated squarer (see Sect. 7.2.4, p. 167).

- The LSB of R_0 is ℓ_0 , the LSB of D is $-w_F$, and therefore, the LSB of the exact product R_0D is $\ell_0 - w_F$. It is also the LSB of $E_0 = 1 - R_0D$ when computed exactly. The format of E_0 , if computed exactly, is therefore ufix($m_0, \ell_0 - w_F$). In Fig. 9.17, we have a parameter ℓ that defines the LSB of both E_i and Q_i . A rather conservative choice would be $\ell = \ell_0 - w_F$. The error analysis below will tell us if it may be larger, which would save hardware. Obviously, E_0 should in any case be computed at least accurately enough so that all the LSBs of D are taken into account.

in Sect. 9.2.8, and in both cases this step is profitable because it makes the problem easier by bringing the dividend closer to 1.

- The bits of E_0 higher than m_0 are known to be zeroes; therefore, the multiplier for $R_0 D$ need not compute any bit higher than m_0 , and the subtracter that computes E_0 operates on $\text{ufix}(m_0, \ell)$ numbers.
- Similarly the format of Q_0 is $\text{ufix}(0, \ell_0 - w_F)$ if computed exactly. Figure 9.17b shows a multiplier that truncates to LSB ℓ , to be determined by the error analysis.
- If computed exactly, the LSBs of E_i and Q_i move further and further to the right. It is therefore desirable to truncate them. It is possible to use the same LSB parameter ℓ for Q_i and E_i and for each iteration.
- In Fig. 9.17b, the computation of Q_{i+1} is implemented as $Q_{i+1} = Q_i + Q_i \times E_i$, which allows for a smaller multiplier than $Q_{i+1} = Q_i \times (1 + E_i)$. Indeed there are many known zeroes in $1 + E_i$, and it would be a waste of hardware to multiply by them. Of course, the latter formulation may still be preferred if the target hardware includes a fixed-width multiplier (e.g., a DSP block on FPGAs) large enough to input $1 + E_i$.
- The correction term $Q_i E_i$ is smaller than 2^{m_i+1} ; therefore, this term is smaller with each iteration. The LSB of E_i is chosen as $\ell - 1$, so as to be no more accurate than needed as input to the multiplier for $Q_i \times E_i$.

To conclude this section, let us sketch an error analysis that will help determine ℓ and the truncation parameters. Following the methodology of Chap. 3, the technique consists of successive rewritings of the definition of the error between the computed result and the expected quotient, inserting relevant intermediate terms:

$$\delta_Q = Q_n - X/D \quad (9.77)$$

$$= \underbrace{Q_n - q_n}_{=\delta_n} + \underbrace{q_n - X/D}_{=\delta_{\text{approx}}} \quad (9.78)$$

Here $\delta_n = Q_n - q_n$ captures all the rounding errors due to transforming equations (9.71)–(9.75) into Fig. 9.17, while δ_{approx} is the error due to truncating the infinite series (9.66) to its n first terms:

$$\delta_{\text{approx}} = q_n - X/D \quad (9.79)$$

$$= X R_0 (1 + e_0) (1 + e_0^2) \cdots (1 + e_0^{2^{n-1}}) - X \frac{R_0}{1 - e_0} \quad (9.80)$$

$$= X R_0 \left((1 + e_0) (1 + e_0^2) \cdots (1 + e_0^{2^{n-1}}) - \frac{1}{1 - e_0} \right) \quad (9.81)$$

$$= X R_0 \left(\sum_{j=2^n}^{\infty} e_0^j \right). \quad (9.82)$$

With $|X R_0| < 2$, a tight bound on δ_{approx} can be computed from the initial bound on e_0 .

Let us now focus on the accumulation of rounding errors. The bound on the overall rounding error on E_i is defined first:

$$\delta'_{i+1} = E_{i+1} - e_{i+1} \quad (9.83)$$

$$= E_{i+1} - \tilde{E}_i^2 + \tilde{E}_i^2 - E_i^2 + E_i^2 - e_i^2 \quad (9.84)$$

$$= \delta_i^{\text{square}} + \delta_i^{\text{trunc}E} + (E_i + e_i)\delta'_i \quad (9.85)$$

with $\delta'_0 = 0$.

A faithful truncated squarer as in Fig. 9.17 will have an error δ_i^{square} bounded by $2^{\ell-1}$. The factor $(E_i + e_i)$ is easy to bound by 2^{m_i+1} .

The truncation before the square is optional; the corresponding error $\delta_i^{\text{trunc}E} = \tilde{E}_i^2 - E_i^2$ may be rewritten $\delta_i^{\text{trunc}E} = (\tilde{E}_i + E_i)(\tilde{E}_i - E_i)$. Again the factor $(\tilde{E}_i + E_i)$ is bounded by 2^{m_i+1} , and by definition of a truncation $|\tilde{E}_i - E_i| < 2^{\ell'_i}$, thus, an error bound on $\delta_i^{\text{trunc}E}$ is $2^{m_i+1+\ell'_i}$. To balance in (9.85) this error with δ_i^{square} (bounded by $2^{\ell-1}$), a sensible choice for ℓ'_i is

$$\ell'_i = \ell - m_i - 2. \quad (9.86)$$

The input size to the squarer is then $m_i - \ell'_i + 1 = 2m_i - \ell + 3 = 1 + m_{i+1} - (\ell - 1) + 1$: another point of view on this truncation is that the input size to the squarer should match its output size (it is one bit larger).

In summary, the recurrence (9.85) leads to the following bound

$$\bar{\delta}'_{i+1} = \begin{cases} 2^\ell + 2^{m_i+1}\bar{\delta}'_i & \text{for steps with truncation as per (9.86)} \\ 2^{\ell-1} + 2^{m_i+1}\bar{\delta}'_i & \text{for steps without truncation} \end{cases} \quad (9.87)$$

with $\bar{\delta}'_0 = 0$.

Since 2^{m_i+1} is much smaller than 1, $\bar{\delta}'_{n-1} \approx 2^\ell$: with all its truncations and roundings, the E_i datapath is almost as accurate as its last step.

To fix ideas, let us take an example: for $\ell = -32$, with $m_0 = -7$, hence $m_1 = -14$ and $m_2 = -28$, we will truncate the first step using (9.86) at $\ell'_0 = -32 + 7 - 2 = -27$, so the first squarer has an ufix($-7, -27$) input (21 bits) and an ufix($-14, -33$) output (20 bits), with $\bar{\delta}'_1 = 2^{-32}$. We truncate the input to the second squarer again at $\ell'_1 = -32 + 14 - 2 = -20$: it has an ufix($-14, -20$) input (7 bits) and an ufix($-28, -33$) output (6 bits). This yields $\bar{\delta}'_2 = 2^{-32} + 2^{-13-32}$.

The take-away message of this example is that the quadratic convergence is here expressed as each square having its input/output size smaller by a number of bits that double at each iteration. In Newton-Raphson, conversely, the datapath size was doubling at each iteration.

Now let us turn to the Q_i datapath. The final fixed-point addition is exact; hence Q_{i+1} may be rewritten as $Q_{i+1} = Q_i + P_i$ where P_i is the multiplier

output (see Fig. 9.17). The error on Q may thus be rewritten as

$$\delta_{i+1} = Q_{i+1} - q_{i+1} \quad (9.88)$$

$$= (Q_i + P_i) - q_i(1 + e_i) \quad (9.89)$$

$$\begin{aligned} &= (Q_i + P_i) - (Q_i + \tilde{Q}_i E_i) \\ &\quad + (Q_i + \tilde{Q}_i E_i) - (Q_i + Q_i E_i) \\ &\quad + Q_i(1 + E_i) - Q_i(1 + e_i) \\ &\quad + Q_i(1 + e_i) - q_i(1 + e_i) \end{aligned} \quad (9.90)$$

$$= \delta_i^{\text{mult}} + \delta_i^{\text{trunc}Q} + Q_i \delta'_i + (1 + e_i) \delta_i . \quad (9.91)$$

Thus, (9.91) is a recurrence that expresses δ_{i+1} as a function of δ_i and δ'_i . Since the factor $(1 + e_i) \approx 1$, the errors due to each step accumulate here (contrary to the E_i datapath where the error from step i was scaled down by 2^{m_i} for the next step). This justifies the use of a common LSB ℓ for all the steps.

Among the four terms of (9.91), we may bound δ_i^{mult} by 2^ℓ (faithful multiplier). Using $Q_i < 2$ and the bound on δ'_i just computed, we may bound $Q_i \delta'_i$ by a value that is slightly larger than $2^{\ell+1}$. For balanced error contributions, we should bound $\delta_i^{\text{trunc}Q} = (\tilde{Q}_i - Q_i)E_i$ by something of the order of 2^ℓ as well. As E_i is bounded by 2^{m_i} , this is achieved by truncating the Q_i multiplier input to

$$\ell''_i = \ell - m_i . \quad (9.92)$$

Again we find that the multiplier should have comparable input and output sizes. With this choice, (9.91) may be rewritten as

$$\bar{\delta}_{i+1} = 2^{\ell+1} + 2\bar{\delta}'_i + (1 + 2^{m_i})\bar{\delta}_i . \quad (9.93)$$

This bound is valid if some steps do not use truncation, although it may be tightened in this case.

With (9.93) combined with the bound on the approximation error δ_{approx} thanks to (9.81), it becomes finally possible to determine the smallest ℓ that will ensure a faithfully rounded divider. There is probably no nice closed-form formulation, but it is easy to write a program that implements these equations.

9.5 Other Equivalent Multiplicative Methods

We now survey other techniques that turn out to be variations of the ones we just detailed.

9.5.1 Multiplicative Normalization (Goldschmidt's Algorithm)

This algorithm defines two parallel recurrences n_i (numerator) and d_i (denominator), with an invariant $\frac{n_i}{d_i} = \frac{1}{D}$, as follows:

$$n_i = 1 \cdot p_0 p_1 \cdot p_2 \cdots p_i \quad (9.94)$$

$$d_i = D \cdot p_0 \cdot p_1 \cdot p_2 \cdots p_i \quad (9.95)$$

with p_i chosen such that $d_i \rightarrow 1$, hence $n_i \rightarrow 1/D$, as follows: defining the usual error $e_i = 1 - n_i D = 1 - d_i$ and imposing on it quadratic convergence (i.e., $e_{i+1} = e_i^2$) yield $p_{j+1} = 2 - d_j$, so the complete recurrence is [EL94]

$$p_0 \approx \frac{1}{D} \quad (9.96)$$

$$d_0 = D p_0 \quad (9.97)$$

$$n_0 = p_0 \quad (\text{or } n_0 = p_0 X \text{ for direct quotient computation}) \quad (9.98)$$

$$n_{j+1} = p_j \times n_j \quad (9.99)$$

$$d_{j+1} = p_j \times d_j \quad (9.100)$$

$$p_{j+1} = 2 - d_j \quad (9.101)$$

Now, $d_i \rightarrow 1$ hence $p_i \rightarrow 1$: the above recurrence involves accurate multiplications by numbers close to one. This is not hardware-efficient: if $d_i \rightarrow 1$, it is better to use the error $e_i = 1 - d_i$ as introduced above and rewrite $p_j \times d_j$ as $p_j + p_j \times e_j$, which involves a smaller multiplier and avoids multiplying by the known zeroes in the binary representation of d_j . Doing this (and rewriting p_i as well) yields the quadratic series of the previous section.

A second textbook method consists of cutting (9.66) at its first term and recomputing e_i out of each reciprocal approximation R_i . This yields yet another recurrence:

$$R_0 \approx \frac{1}{D} \quad (9.102)$$

$$e_i = 1 - R_i \times D \quad (9.103)$$

$$R_{i+1} = R_i + R_i \times e_i \quad (9.104)$$

which is a decomposition in two steps of our first Newton-Raphson recurrence (9.53).

9.5.2 Higher-Order Householder Methods

To improve an approximation x_i of the root of a function f (here $f(x) = 1/x - D$), Newton's method locally approximates f by its tangent: this linear or degree-1 approximation involves only $f(x_i)$ and $f'(x_i)$. The Householder method of order d is a generalization to a degree- d local approximation to the function, built using derivatives up to the d -th one. With a higher degree, the approximation is closer to the function; therefore, x_{i+1} is closer to the actual root: convergence is faster. In general, in an order- d Householder method, the number of correct bits in x_i is multiplied by $d + 1$ at each iteration. However, higher-degree Householder approximations also come at a higher cost in terms of evaluation.

The order-two Householder method is also known as Halley's method. When applied to the reciprocal, the method must be slightly adapted [IP17] to avoid obtaining $x_{i+1} = 1/D$ [Fly70]. This yields the formula

$$x_{i+1} = x_i(1 + e_i + e_i^2) \quad \text{with} \quad e_i = 1 - Dx_i \quad (9.105)$$

where we again recognize (9.65), p. 294, this time truncated to three terms. In (9.105), e_i has to be recomputed at each iteration.

Thanks to cubic convergence, a single Halley iteration triples the precision of R_0 at the cost of one squarer and two multipliers. This fills a gap between one iteration of the reciprocal-only variant of Fig. 9.17 (doubling the precision of R_0 at the cost of two multipliers) and two iterations of the same architecture (quadrupling the precision of R_0 using one squarer and 3 multipliers). This approach has been shown [IP17] to be relevant on FPGAs for precisions up to 32 bits: starting with a table with 11 input bits provides a 32-bit faithful reciprocal.

9.5.3 Ad hoc Evaluation of Series Expansion

The VFloat library [WBL06] uses yet another variation on the series expansion: it splits D between its higher and lower bits $D = D_H + D_L$ with $D_L < 2^k$ for k some negative integer. Then,

$$\frac{X}{D} = \frac{X}{D_H} \cdot \frac{1}{(1 + D_L/D_H)}, \quad (9.106)$$

using $\frac{1}{1+x} \approx 1 - x$ (the first elements of its Taylor expansion), we get

$$\frac{X}{D} = \frac{X}{D_H} \cdot (1 - D_L/D_H + \delta_{\text{approx}}) \quad (9.107)$$

$$= X \cdot (D_H - D_L) \cdot \frac{1}{D_H^2} + X\delta_{\text{approx}}. \quad (9.108)$$

The architecture consists of tabulating $\frac{1}{D_H^2}$ and evaluating (9.108) as $(X \times (D_H - D_L)) \times \left(\frac{1}{D_H^2}\right)$, so the first multiplication and the table read can execute in parallel.

Let us give hand-waving elements of error analysis. For a result accurate to precision ℓ , the approximation error must be smaller than 2^ℓ . As $\delta_{\text{approx}} = (D_L/D_H)^2 - (D_L/D_H)^3 \dots$ (the remaining elements of the Taylor expansion), the parameter k must be chosen so that $2k < \ell$. Furthermore, $\frac{1}{D_H^2}$ must be tabulated to an accuracy of at least 2^ℓ : the table has about twice as many output bits as input bits. Finally, the two multipliers must be faithful to a few bits more than 2^ℓ , and their inputs are slightly larger than full significands: in general, the overall cost is larger than one-step versions of Newton-Raphson or quadratic series expansion. However, this technique is well suited to small precisions on FPGAs when the multipliers fit in DSP blocks.

The same decomposition as $D = D_H + D_L$ with $D_L < 2^k$ can be used for larger precisions, but then more terms of the series expansion must be considered [Jai+14]. In this case the formula used is

$$\frac{1}{D} = \frac{1}{D_H} \cdot \frac{1}{(1 + D_L/D_H)} \quad (9.109)$$

$$= \frac{1}{D_H} (1 - e + e^2 - e^3 + \dots) \quad \text{with } e = D_L/D_H \quad (9.110)$$

The idea is then to tabulate $r_0 = 1/D_H$, choosing k such that D_H fits on few bits. Then e can be rewritten $e = (D - D_H)/D_H = D/D_H - 1 = r_0 D - 1$: we see that (9.110) is exactly (9.65) which was the basis of the previous section. With some effort to implement (9.110) so that it computes just right [Jai+14], the methods are really equivalent. However, [Jai+14] tabulates r_0 as a full-precision number and then computes e as the (full-size) product $r_0 D_L$. This is more expensive than the low-precision tabulation of r_0 used in Sect. 9.4.

9.6 Square Root, the Little Sister of Division

Division and square root are deeply related, and the corresponding algorithms are very close one to the other. An intuition to explain this is that the two problem formulations are very close. Euclidean division is defined

as follows: *Given two integer X and D, find two integers Q and R such that $X = QD + R$ and $0 \leq R < D$.* The equivalent integer square root formulation is *Given an integer X, find an integer S such that $S^2 \leq X < (S + 1)^2$.* Using $(S + 1)^2 = S^2 + 2S + 1$, this formulation can be rephrased as *Given an integer X, find an integer S such that $X = S^2 + R$ and $0 \leq R < 2S + 1$.* This second formulation is very close to the formulation of Euclidean division. Due to this similarity, algorithms for square root are similar to algorithms for division. Digit-recurrence algorithms for square root [EL94; Par10; LM99] will be reviewed and generalized in Chap. 19. There also exists multiplier-based approaches, including series expansion [WBL06] or a Newton-Raphson iteration that converges to $1/\sqrt{X}$ [EL04; Par10]. The function to zero is then $f(x) = 1/x^2 - X$. This leads to the following recurrence:

$$s_{i+1} = \frac{1}{2}s_i(3 - s_i^2 X) \quad . \quad (9.111)$$

A variant of Goldschmidt algorithm for square root also exists. The iterations for division and square root can even share most of the hardware [PB02].

In addition, since square root is a unary function, there is also the possibility to use any of the generic function approximation techniques that will be presented in Part III, for instance, polynomial approximation [Din+10], which may also be used to provide the initial approximation to a multiplicative iteration [PB02; Pas12].

References

- [Bru18] Javier D. Bruguera. "Radix-64 Floating-Point Divider". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2018, pp. 87–94 (cit. on pp. 4, 288, 349, 350).
- [CHT02] Marius Cornea, John Harrison, and Ping Tak Peter Tang. *Scientific Computing on Itanium® -Based Systems*. Intel Press, 2002 (cit. on pp. 2, 294, 347).
- [Din+10] Florent de Dinechin, Mioara Joldes, Bogdan Pasca, and Guillaume Revy. "Multiplicative square root algorithms for FPGAs". In: *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2010, pp. 574–577 (cit. on pp. 13, 303, 307, 356).
- [DM95] Debjit Das Sarma and David W. Matula. "Faithful Bipartite ROM Reciprocal Tables". In: *12th Symposium on Computer Arithmetic*. Ed. by S. Knowles and W.H. McAllister. IEEE, 1995, pp. 17–28 (cit. on p. 292).

- [EL04] Miloš D. Ercegovac and Tomás Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004 (cit. on pp. 3, 11, 23, 214, 218, 259, 267, 303).
- [EL90] Miloš D. Ercegovac and Tomás Lang. “Simple Radix-4 Division with Operands Scaling”. In: *IEEE Transactions on Computers* 39.9 (1990), pp. 1204–1208 (cit. on pp. 283, 285).
- [EL94] Miloš D. Ercegovac and Tomás Lang. *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, Boston, 1994 (cit. on pp. 3, 259, 300, 303).
- [ELM94] Miloš D. Ercegovac, Tomás Lang, and Paolo Montuschi. “Very-High Radix Division with Prescaling and Selection by Rounding”. In: *IEEE Transactions on Computers* 43.8 (1994), pp. 909–18 (cit. on p. 285).
- [Fly70] Michael J. Flynn. “On Division by Functional Iteration”. In: *IEEE Transactions on Computers* C-19.8 (1970), pp. 702–706 (cit. on p. 301).
- [GES07] Ronen Goldberg, Guy Even, and Peter M. Seidel. “An FPGA Implementation of Pipelined Multiplicative Division With IEEE Rounding”. In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2007, pp. 185–196 (cit. on p. 288).
- [HOH97] David L. Harris, Stuart F. Oberman, and Mark A. Horowitz. “SRT Division Architectures and Implementations”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 1997, pp. 18–25 (cit. on p. 287).
- [IP17] Matei Iștoan and Bogdan Pasca. “Flexible Fixed-Point Function Generation for FPGAs”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2017, pp. 123–130 (cit. on pp. 13, 260, 301).
- [Jai+14] Manish Kumar Jaiswal, Ray C.C. Cheung, M. Balakrishnan, and Kolin Paul. “Series Expansion based Efficient Architectures for Double Precision Floating Point Division”. In: *Circuits, Systems, and Signal Processing* 33 (2014), pp. 3499–3526 (cit. on pp. 260, 288, 302).
- [LCC06] Jianhua Liu, Michael Chang, and Chung-Kuan Cheng. “An Iterative Division Algorithm for FPGAs”. In: *International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM, 2006, pp. 83–89 (cit. on p. 285).
- [LM99] Tomás Lang and Paolo Montuschi. “Very High Radix Square Root with Prescaling and Rounding and a Combined Division/Square Root Unit”. In: *IEEE Transactions on Computers* 48.8 (1999), pp. 827–841 (cit. on p. 303).
- [Mar00] Peter Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000 (cit. on pp. 4, 24, 260, 288, 347).

- [Obe99] Stuart F. Oberman. "Floating Point Division and Square Root Algorithms and Implementation in the AMD-K7 Microprocessor". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 1999, pp. 106–115 (cit. on pp. 260, 288).
- [OF97a] Stuart F. Oberman and Michael J. Flynn. "Design Issues in Division and Other Floating-Point Operations". In: *IEEE Transactions on Computers* 46.2 (1997), pp. 154–161 (cit. on pp. 3, 4, 260).
- [OF97b] Stuart F. Oberman and Michael J. Flynn. "Division Algorithms and Implementations". In: *IEEE Transactions on Computers* 46.8 (1997), pp. 833–854 (cit. on pp. 259, 260, 288).
- [Par10] Behrooz Parhami. *Computer Arithmetic, Algorithms and Hardware Designs*. 2nd ed. Oxford University Press, 2010 (cit. on pp. 23, 267, 287, 303).
- [Pas12] Bogdan Pasca. "Correctly Rounded Floating-Point Division For DSP-Enabled FPGAs". In: *Field Programmable Logic and Applications*. 2012 (cit. on pp. 13, 260, 288, 291, 292, 303).
- [PB02] José-Alejandro Piñeiro and Javier D. Bruguera. "High-Speed Double-Precision Computation of Reciprocal, Division, Square Root, and Inverse Square Root". In: *IEEE Transactions on Computers* 51.12 (2002), pp. 1377–1388 (cit. on pp. 13, 260, 288, 303).
- [Rob58] James E. Robertson. "A New Class of Digital Division Methods". In: *IRE transactions on electronic computers* 3 (1958), pp. 218–222 (cit. on p. 272).
- [SBD04] Gustavo Sutter, Gery Bioul, and Jean-Pierre Deschamps. "Comparative Study of SRT-Dividers in FPGA". In: *International Conference on Field-Programmable Logic and Applications (FPL)*. Springer, 2004, pp. 209–220 (cit. on p. 278).
- [SL97] Peter Soderquist and Miriam Leeser. "Division and Square Root: Choosing the Right Implementation". In: *IEEE Micro* 17.4 (1997), pp. 56–66 (cit. on pp. 260, 288).
- [Sun+84] David A. Sunderland, Roger A. Strauch, Steven S. Wharfield, Henry T. Peterson, and Christopher R. Role. "CMOS/SOS Frequency Synthesizer LSI Circuit for Spread Spectrum Communications". In: *IEEE Journal of Solid-State Circuits* 19.4 (1984), pp. 497–506 (cit. on p. 292).
- [Toc58] Keith D. Tocher. "Techniques of Multiplication and Division for Automatic Binary Computers". In: *The Quarterly Journal of Mechanics and Applied Mathematics* 11.3 (1958), pp. 364–384 (cit. on p. 272).
- [WBL06] Xiaojun Wang, Sherman Braganza, and Miriam Leeser. "Advanced Components in the Variable Precision Floating-Point Library." In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2006, pp. 249–258 (cit. on pp. 13, 260, 288, 301, 303).



10

CHAPTER 10

Shifters and Leading Bit Counters

A shifter is a digital circuit that has two inputs and one output. It shifts one of the inputs by a number of bits defined by the second input. Although not strictly speaking arithmetic components themselves, shifters are pervasive in application-specific arithmetic. Their main use is to multiply by a power of two, to convert fixed-point to floating-point or the other way around, to align the binary representations of two numbers before adding them, and to normalize a floating-point result. In the latter case, the shift amount is the result of a leading bit count. This chapter covers all these use cases, studying their requirements and proposing relevant architectures.

All the general-purpose processors offer a handful of shift-related instructions, but they also hide many more shifters of different sizes and shapes in their floating-point units. In the present book, shifters are used in Chaps. 11, 18, 21, and 22. They are also useful in the fixed-point implementation of some elementary functions such as square root [Din+10] or arctangent [DI15], among others. The requirements of all these applications in terms of shifters are quite varied, and the shifters are often closely associated with other functionality, such as leading zero counting (to determine the shift value) and “sticky” bit computation (to compress the information from the shifted-out bits into a single bit that can be used to decide on the rounding direction). The purpose of this chapter is to expose this diverse landscape in the context of application-specific arithmetic.

10.1 Shifters

A shifter, as depicted in Fig. 10.1, inputs a bit vector X of w_X bits and a shift value S on w_S bits and shifts X by S bits. From an arithmetic point of view, shifters are used to multiply or divide a binary number X by a power of two 2^S .

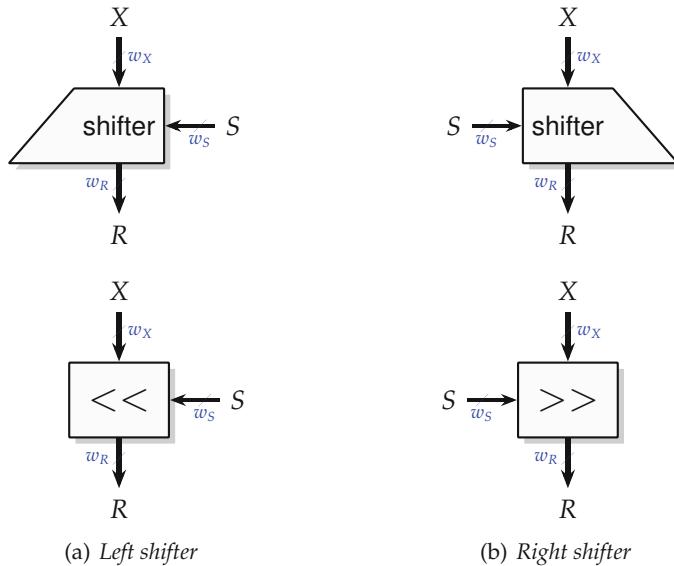


Fig. 10.1 Classical graphical representations of shifters. X is the bit vector to be shifted, and S is the shift value.

There are left shifters and right shifters, which are perfectly symmetrical: the architecture of a right shifter can always be deduced from that of a left shifter by mirror image.

10.1.1 Avoiding Bidirectional Shifters

One could think that a bidirectional shifter may sometimes be useful. Such a component would either input a direction bit (left or right) or (almost equivalently¹) a signed shift distance S .

However, in most cases, the proper way to address a problem where a given data may be either shifted left or right is to consider that it will be shifted left only from its rightmost possible position. The architecture overhead is the addition of a constant offset (the largest possible right shift) to the signed shift distance S , converting it to an unsigned one. Usually, this addition is for free. For instance, in a floating-point context, a signed shift distance is related to a signed exponent which is already biased by some constant (see Chap. 2); therefore the constant offset can be merged in the constant bias subtraction. See Chap. 22 for a detailed example. In any case, the

¹ A direction bit and a positive shift together represent a signed shift as sign+magnitude.

shift distance is always coded on a very small number of bits (e.g., $w_S \leq 6$ for shifts up to 64 bits); therefore an addition to compute S will be cheap.

Reducing all shifts to unidirectional ones usually leads to more efficient architectures, in particular in the general case where the maximum left and right shift distances were not identical.

Therefore, in this chapter and throughout this book, all shifters are unidirectional. There are left shifters and right shifters, but the direction is an input to the shifter generator (Fig. 10.2), not to the shifter itself (Fig. 10.1).

10.1.2 Parameters of a Generic Shifter Generator

The generic interface to a shifter generator is illustrated in Fig. 10.2. The parameters are the direction (left or right), the input size w_X , the maximum shift distance S_{\max} , and the output size w_R .

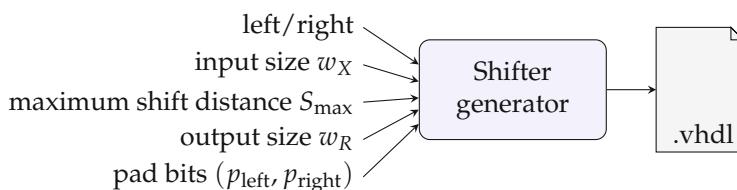


Fig. 10.2 Interface to a generic shifter generator.

As Fig. 10.3 shows, the output R of a shifter will include bits of the input X but also left and right pad bits (represented as dots on this figure). The last parameter ($p_{\text{left}}, p_{\text{right}}$) describes what these pad bits should be. There are several ways to express this information; here is the one used in the FloPoCo interface: for each of p_{left} and p_{right} , a value of 0 means that the shifter should pad with 0; a value of 1 means that the shifter should pad with 1; a value of -1 means that the shifter architecture will have an input bit (in addition to the input words X and S shown in Fig. 10.1) defining at runtime the pad value. This last case is essentially used for two's complement arithmetic, which often requires left-padding with the sign bit of the input (sign extension; see Sect. 2.2.3). Another point of view is that the division of a signed fixed-point number X by a power of two 2^S can be implemented by right-shifting X by S bits while left-padding with the sign bit of X . Relatedly, microprocessors often offer two right-shift instructions, typically called LSR for Logical Shift Right (left-padding with 0) and ASR for Arithmetic Shift Right (left-padding with the sign bit).

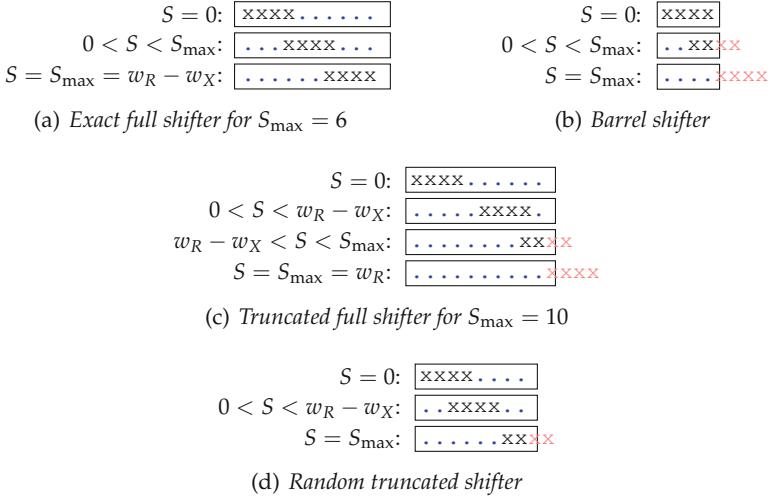


Fig. 10.3 Some outputs of various right shifters for an input size $w_X = 4$ bits. Bits coming from X are marked x , pad bits are blue dots, and discarded bits (if any) are in red.

In FloPoCo, the information $(p_{\text{left}}, p_{\text{right}})$ is optional and defaults to $(0, 0)$, and the remainder of this chapter will mostly use zero padding for simplicity.

Let us now discuss the other parameters visible in Fig. 10.1 and the constraints between all these parameters. There are actually several classes of shifters, and each is useful in some context.

An *exact full shifter* is such that the output size is $w_R = w_X + S_{\max}$. In this case, whatever the value of S (with $0 \leq S \leq S_{\max}$), no bit of X is shifted out of R (see Fig. 10.3a).

It is however often desirable to design a shifter such that $w_R \leq w_X + S_{\max}$, in which case bits will be discarded by the shifter for some values of S (see Fig. 10.3b, c, and d). Such a shifter is called a *truncated shifter*.² The discarded bits do not necessarily correspond to lost information. For instance, consider a left shifter used to normalize a floating-point number, i. e., bring its leading non-zero bit at the most significant bit position. It may discard all the bits to the left of this MSB position, since the discarded bits are, by definition, all zeroes: no information is lost. Another common case is a right shifter where the discarded bits entail an error small enough considering the target accuracy (see Chap. 22 for an example).

² Maybe *truncating* shifters would be more accurate: strictly speaking, it is the output that is truncated. The term “truncated” is chosen here to match the “truncated multipliers” of multiplier literature. In both cases, the motivation is architectural savings.

An option to build a truncated shifter is use an exact full shifter and then discard some of the bits of its output. However, in this option, there will be hardware that only shifts bits doomed to be discarded. The objective of a truncated shifter is to save this hardware.

There are two extremal cases of truncated shifters. A *truncated full shifter* may completely shift out X (see Fig. 10.3c). It is characterized by the relationship $S_{\max} = w_R$. A *barrel shifter* is characterized by $w_X = w_R$ and $S_{\max} = w_X$. It is the shifter used inside the integer arithmetic and logic unit (ALU) of a microprocessor,³ and it is necessarily truncated (see Fig. 10.3b).

A shifter generator restricted to any of the previous classes could require fewer parameters, since each class is defined by an equation between the parameters (e.g., $w_X = w_R$ for barrel shifters). However, application-specific arithmetic has a use for all these shifter classes and even often resorts to shifters that belong to none of them (see Fig. 10.3d), hence the need for all the parameters in Fig. 10.2.

Still, we may state some commonsense constraint on these parameters.

- When $w_R > w_X$, it makes no sense to have $w_R > w_X + S_{\max}$: some of the output bits could never be reached by bits of X . All the bits are shifted out as soon as $S \geq w_R$, so it also makes no sense to have $S_{\max} > w_R$. The space of useful shifters is therefore characterized in this case by $S_{\max} \leq w_R \leq w_X + S_{\max}$.
- When $w_R < w_X$, it makes no sense to have $w_X > w_R + S_{\max}$: it would mean that some bits of X can never appear in R . Also, all the bits are shifted out as soon as $S \geq w_X$, so it also makes no sense to have $S_{\max} > w_X$. The space of useful shifters is therefore characterized in this case by $S_{\max} \leq w_X \leq w_R + S_{\max}$.

A single constraint that captures the space of useful shifters is therefore

$$S_{\max} \leq \max(w_R, w_X) \leq \min(w_R, w_X) + S_{\max} . \quad (10.1)$$

One last parameter which is not given in Fig. 10.2 is the size w_S of the shift input S . It is simply defined out of S_{\max} as $w_S = \lceil \log_2(S_{\max} + 1) \rceil$.

Note how it would be a bad idea to have w_S as parameter and deduce the maximum shift distance as $S_{\max} = 2^{w_S} - 1$. All the previous constraints were expressed in terms of S_{\max} , and it would be too restrictive to be limited to S_{\max} values of the form $2^{w_S} - 1$. For instance, a floating-point adder for one of the standard IEEE 754 formats needs two shifters, each with a maximum shift distance of $w_F + 2$ bits where w_F is the size of the significand fraction [Mul+18]. For the standard IEEE 754 formats, this corresponds to maximum shifts of 12, 25, or 54 bits: none of these is of the form $2^{w_S} - 1$.

³ In a microprocessor, w_X is usually a power of two, and it might be cheaper to limit S_{\max} to $w_X - 1$ and to manage specifically the case $S_{\max} = w_X$, which completely shifts out the input.

Besides, it is easier to think of arithmetic problems in terms of maximum shift distance than in terms of its size in bits.

10.1.3 Architecture of an Exact Full Shifter

A basic shifter architecture consists of a sequence of multiplexers shifting by increasing powers of two, as shown in Fig. 10.4. The simplest architecture is derived for an exact full shifter when $S_{\max} = 2^{w_S} - 1$ as shown in this figure.

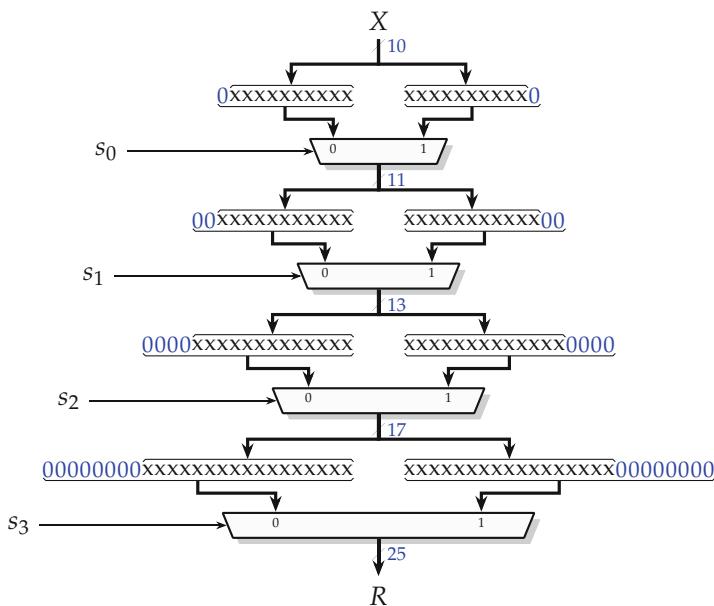


Fig. 10.4 Architecture of a full right shifter for $w_X = 10$ and $S_{\max} = 15$. The s_i are the bits of S . For a left shifter, just exchange the two inputs to each multiplexer, or (equivalently) complement all the s_i .

Hands on: Shifters in FloPoCo

The following command produces the shifter of Fig. 10.4:

```
flopoco Shifter wX=10 maxShift=15 dir=1
```

This architecture consumes the bits of S in an least significant bit (LSB) first manner. Note that it is possible to build a shifter for any order of the multiplexers, but the order presented in Fig. 10.4 is arguably the best for a full shifter:

- Firstly, it minimizes the sum of the multiplexer sizes. For instance, in the figure ($w_X = 8$ and $w_S = 4$), the number of elementary 1-bit muxes is $(w_X + 1) + (w_X + 3) + (w_X + 7) + (w_X + 15)$. If the bits of S were consumed most significant bit (MSB) first, the first multiplexer would produce a $(w_X + 8)$ -bit intermediate result, and the total size would be $(w_X + 8) + (w_X + 12) + (w_X + 14) + (w_X + 15)$.
- Secondly, it often happens that the bits of S come from an adder or subtractor. In this case, the carry propagation produces the lower bits first. The LSB-first arrangement shown in Fig. 10.4 thus also optimizes the overall latency by allowing an overlap between the computation of S and the shift process. It indirectly saves on area as well, since a fast adder is not needed to compute S .

In Fig. 10.4, the input word is padded left and right with zeroes. This simplifies the corresponding MUXEs to AND gates [Che+18]. In some situa-

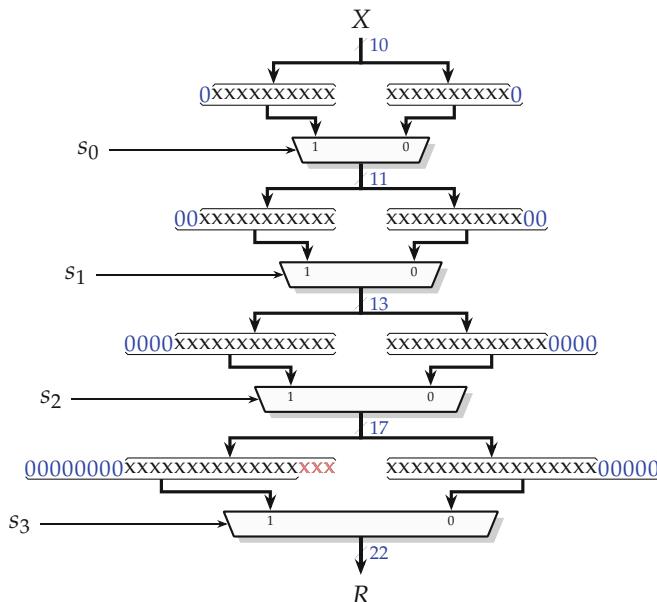


Fig. 10.5 An example of exact full shifter (to the right) when S_{\max} is not of the form $2^{w_S} - 1$ (here $S_{\max} = 12$ for $w_X = 10$). Only the last stage is not full. As this shifter is exact, the discarded red bits in the last stage can only be zeroes introduced by previous stages, for instance, when $S = 8 = 1000_2$ or $S = 12 = 1100_2$.

tions, it is desirable to pad with ones or to pad with a bit provided as another input to the shifter. For instance, to multiply by a power of two a number X written in two's complement, a shifter should pad left with the sign bit of X (its MSB).

For an exact full shifter when S_{\max} is not of the form $2^{ws} - 1$ (here $S_{\max} = 12$), the architecture is identical with the exception of the last stage, as illustrated in Fig. 10.5.

FPGA-specific remark

In modern FPGAs with large look-up tables (LUTs), several stages of multiplexing can be merged in one row of LUTs. For instance, a 6-input LUT can be configured as a 4-to-1 multiplexer; therefore two successive rows of Figs. 10.4 or 10.5 can be merged in a single row of 6-input LUTs.

This optimization is discovered by the back-end tools: FloPoCo, for instance, outputs VHDL describing 2-to-1 multiplexers as per Figs. 10.4 and 10.5. However, when building pipelined shifters, it will place pipeline registers every other level.

10.1.4 Barrel Shifter

A barrel shifter (or rectangular shifter) is defined by $w_R = w_X$.

In this case, the order of the shift stages is less relevant since they all have the same size, and all the s_i have the same fanout. The order presented in Fig. 10.6 is still usually preferred for the timing overlap it allows between an addition computing S and the shift.

10.1.5 Barrel Shifter with Early Sticky Bit Computation

In floating-point addition with exponent size w_E and fraction size w_F , the significand of the smaller input must be right-shifted to align it to the other significand (this topic will be discussed in detail in Sect. 11.1).

The actual significand addition will only take place on the upper bits of the shifted significand. However, the lower bits must not be completely discarded: it is important to remember if they were all zeroes, as this information is needed to implement the tie case of correct rounding [Mul+18]. For this purpose, the logical OR of the lower bits must be computed. The result of this logical OR is called the sticky bit for historical reasons (from an age when shifts were performed one bit at a time).

A naive architecture would use a full exact shifter and then a wide OR operation on the lower bits. However, as the OR operation is associative, it

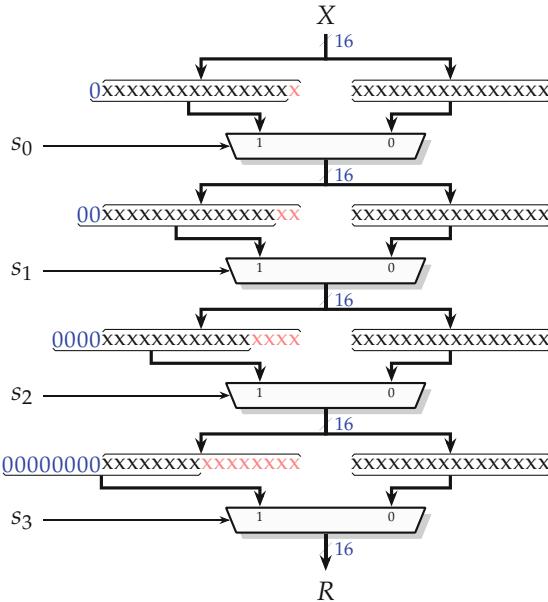


Fig. 10.6 Architecture of a 16-bit right barrel shifter with $S_{\max} = 15$. Red bits are discarded.

is possible to compute the OR incrementally within the shifter, thus discarding the corresponding bits instead of having to shift them further. In other words, it is possible to use the architecture in Fig. 10.6, where the red bits are not discarded but compressed in a tree of OR gates. Overall, this turns a full exact shifter into a barrel shifter.

FPGA-specific remark

A wide-OR operation is not implemented as a tree of OR gate on FPGAs: it is possible to use the fast carry logic for this purpose. Synthesis tools take care of that so that a designer does not have to.

There is an additional possible improvement to this idea. With the stage ordering of Fig. 10.6, the last stage produces the most bits to be OR-related. Reversing the stage order is better in this case. This way, the first stage produces the most bits to send to the OR tree, and the bulk of the sticky computation can be overlapped with the shift. Figure 10.7 shows such a barrel shifter with incremental sticky computation. The parameters for this figure correspond to the shifter in a binary16 floating-point adder. This shifter replaces the exact full shifter in Fig. 10.5 (drawn for the same floating-point setup).

In experiments by the authors, using a shifter with incremental sticky bit computation in reverse order entails a reduction of 20% of the area and 5% of the delay of a single-precision floating-point adder over the use of an exact full shifter followed by a wide OR operation. The area savings essentially come from the fact that a full exact shifter with $w_X = w_F + 1$ and $S_{\max} = w_F + 3$ is replaced with a barrel shifter with $w_X = w_R = w_F + 3$. The delay savings come from the earlier computation of the sticky bit. Comparable savings have been observed for posit addition [UFD19].

10.2 Variations on Leading Zero Counters

A leading zero counter (LZC) (also called a leading one detector (LOD)) inputs a bit vector X and determines the position of its leading (leftmost) non-zero bit, which it outputs encoded as a binary integer C (see Fig. 10.8).

Several trivial generalizations are possible: the leading one counter (LOC) determines the position of the first zero, and the leading bit counter (LBC) also inputs a bit b that provides the value of the bit to be counted. Although the LBC generalizes the two others, it has a slightly higher architectural cost due to the fanout of b ; therefore it is better to distinguish these three components.

In the remainder of this section, we focus on the leading zero counter (LZC) for clarity, but most of it can be generalized to LOC and LBC.

The LZC is related to shifters in two ways:

- Its most naive architecture (which despite its naiveness is relevant for FPGAs) is very similar to a shifter and actually internally performs a shift.
- It is often used in conjunction with a left shifter to normalize a number, i.e., bring the leading bit to the left. This is a core component of most floating-point operators, but it is also widely used in elementary function evaluation, when magnifying small inputs will lead to increased accuracy, in particular for functions with singularities at 0 such as \sqrt{x} or $\text{atan}(x/y)$ [DI15].

In this case, as in the sticky computation of the previous section, it will often be beneficial to merge the leading bit computation within the shifter architecture. This combination, called a *normalizer*, is studied in Sect. 10.3.

Let us first focus on the naive architecture.

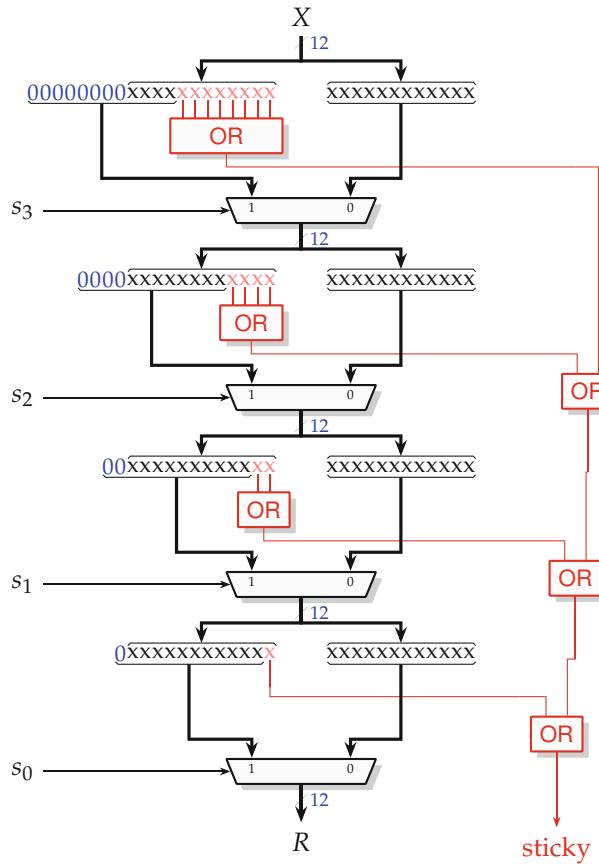


Fig. 10.7 Architecture of a 12-bit right barrel shifter with early sticky bit computation.

Hands on: Combined shifter+sticky in FloPoCo

The following command produces the shifter on the above figure:

```
flopoco Shifter wX=12 maxShift=12 wR=12 \
    dir=1 computeSticky=1
```

10.2.1 Naive LZC Architecture

Let us first assume that the size w_X of the input bit vector X is of the form $w_X = 2^k - 1$. The leading zero count C is an integer between 0 and w_X : it can be represented exactly in binary on k bits, hence $w_C = k$. We call such a component $\text{LZC}(k)$.

The leading bit c_{k-1} of the count C will be 1 iff $C \geq 2^{k-1}$: it can be computed by a NOR of the 2^{k-1} upper bits of X . We then have two cases, illustrated in Fig. 10.9:

- As seen in the first row of Fig. 10.9, if all these bits are 0, then the NOR will be equal to 1, hence $c_{k-1} = 1$. We then have to count the leading zero bits of the remaining $2^{k-1} - 1$ lower bits of X and add this count to $2^{k-1}c_{k-1}$ (this addition will be by concatenation).
- In the other case, there is a non-zero bit in the upper 2^{k-1} bits: we can therefore discard the lower bits and proceed with a leading zero count in the upper bits. As illustrated in Fig. 10.9, we can even discard the middle bit of X : the lower bits of C will be the LZC of the upper $2^{k-1} - 1$ bits of X .

Therefore, the naive architecture of a $\text{LZC}(k)$ consists of a 2^{k-1} -wide NOR controlling a $(2^{k-1} - 1)$ -wide multiplexer choosing between the upper and lower $2^{k-1} - 1$ bits of X . The output of this multiplexer feeds a $\text{LZC}(k-1)$. We have a simple recursive construction that stops for $k = 1$, or $w_X = 1$, where the NOR is no longer useful. This architecture is illustrated in Fig. 10.10.

If w_X is not of the form $w_X = 2^k - 1$, then the count size will be $w_C = \lceil \log_2(w_X + 1) \rceil$. X must simply be padded right with ones to reach the size

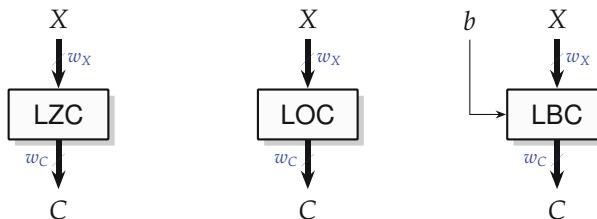


Fig. 10.8 Interfaces to a leading zero counter (left), a leading one counter (middle), and a leading bit counter (right).

0000xxx : $c_2 = 1, c_1c_0$ will be the LZC in xxx
 yyy1xxx : $c_2 = 0, c_1c_0$ will be the LZC in yyy

Fig. 10.9 Cases of leading zero count for $w_X = 7$ and $k = 3$.

$2^{w_C} - 1$, and then a $\text{LZC}(w_C)$ can be used. Padding with ones ensures that C will saturate to w_X when X is all zeroes.

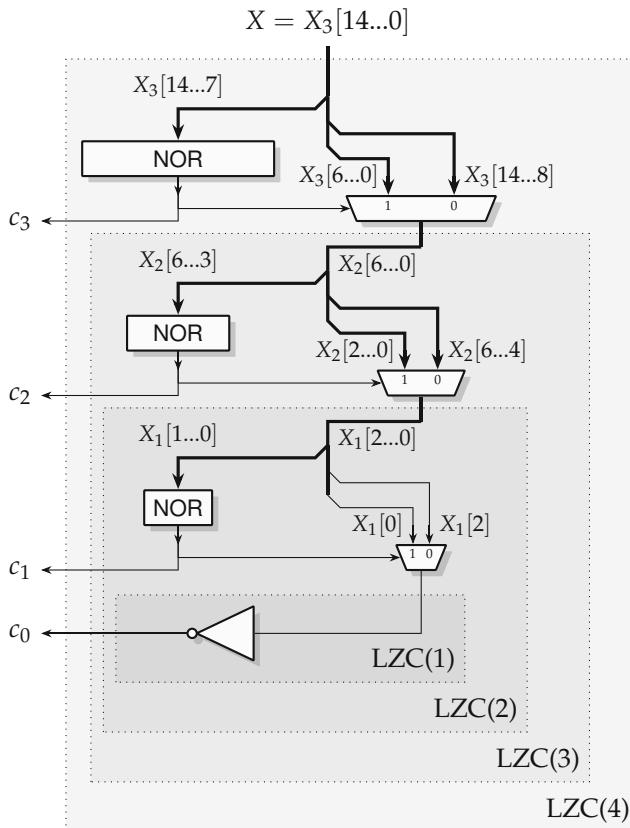


Fig. 10.10 Naive LZC architecture for $w_X = 15$, hence $w_C = 4$.

As seen in Fig. 10.10, the total amount of 1-bit multiplexers is $\sum_{k=1}^{w_C-1} (2^k - 1) = 2^{w_C} - 1 - w_C = w_X - w_C$. This is also the total number of two-input gates to realize all the wide NOR computations. In summary, the area of the naive LZC architecture grows linearly with w_X , just like that of a carry-propagate adder on w_X bits.

To evaluate the delay d_k of $\text{LZC}(k)$, let us assume a binary tree of two-input gates computing the wide NOR. In the first stage of $\text{LZC}(k)$, this tree has depth (hence delay, in gate delay) $k - 1$, to which we must add 1 for the delay of the MUX.⁴ Therefore, d_k verifies the recurrence $d_k = k + d_{k-1}$, hence

⁴ For simplicity, we count a MUX as a single gate delay as we are interested in its asymptotic behavior.

$d_k = k(k+1)/2$ elementary gate delays: the delay of the naive architecture is in $\mathcal{O}(w_C^2)$ or in other words in $\mathcal{O}((\log_2 w_X)^2)$, where \mathcal{O} denotes the Landau symbol (the “big-O” notation) [GKP94].

There exists architecture with a better asymptotic delay: $\mathcal{O}(\log w_X)$. Before presenting them in Sect. 10.2.2, let us stress that the naive architecture is the one to be preferred on FPGAs, as the wide NOR computation is very efficiently implemented by a combination of wide LUTs and fast carry. It also has the advantage that it can be efficiently merged with a shifter, as we will discuss in Sect. 10.3.

10.2.2 Logarithmic-Time LZC Architectures

There are at least two approaches to LZC architectures with true logarithmic time.

10.2.2.1 Oklobdzija’s Recursive Construction

The first is a recursive construction described by Oklobdzija [Okl94], where the architecture of $\text{LZC}(k+1)$ assembles two $\text{LZC}(k)$ and some constant-time logic as illustrated in Fig. 10.11. Here, the block $\text{LZC}(k)$ computes a count C on k bits but also a “zero” bit z , which is equal to 1 iff all the inputs are 0. Using the indices z and C for the left and right components, the equations defining C and v for the $\text{LZC}(k+1)$ are

$$z = z_L \wedge z_R \quad (10.2)$$

$$C = 0C_L \text{ when } z_L = 0 \quad \text{else} \quad 1C_R, \quad (10.3)$$

where $0C_L$ (resp. $1C_R$) denotes the concatenation of a 0 (resp. 1) as the MSB of C_L (resp. C_R), extending these k -bit vectors into $(k+1)$ -bit ones. Another interpretation of z is that for the left part, $z = 1$ means that the count C_L is invalid. The recursion stops at $\text{LZC}(1)$ or $\text{LZC}(2)$, which can be implemented as a handful of gates. Figure 10.11 also gives the truth table for $\text{LZC}(1)$.

Remark that in this construction, the input size of $\text{LZC}(k)$ is $w_X = 2^k$, instead of $w_X = 2^{k-1}$ for the naive LZC architecture. Indeed, on the k bits of C , it is only possible to count from 0 to $2^k - 1$, but we have the output bit z that flags the count value 2^k . With the truth table for $\text{LZC}(1)$ given in Fig. 10.11, the architecture will output $C = 0$ in the case when all the inputs are 0; therefore the z bit of $\text{LZC}(k)$ can also be interpreted as c_k .

The area still grows linearly in w_X : indeed, calling m_k the number of bit-level multiplexers in $\text{LZC}(k)$, we observe that m_k is defined by the recursion $m_{k+1} = k + 1 + 2m_k$, with $m_1 = 1$. This yields $m_k = 2^{k+1} - k - 2$ (the reader is invited to check this result by recurrence). In terms of w_X , we have slightly

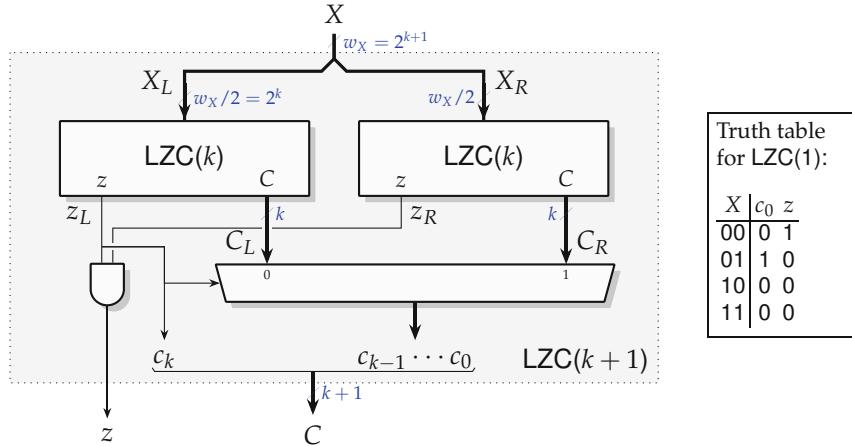


Fig. 10.11 Oklobdzija’s recursive LZC construction.

less than $2w_X$ multiplexers (this is almost twice the MUX count in Fig. 10.10), and then there is a binary tree of w_X AND gates (matching the combined cost of the wide NORs in Fig. 10.10).

In terms of logic delay, the wide multiplexer counts for one gate delay (it is composed of bit-level multiplexers operating in parallel); therefore each level only adds one gate delay to the computation, hence a total delay in $\mathcal{O}(\log_2 w_X)$. However, there remains a large fanout for the multiplexer input, which for large sizes should be accounted for in the delay.

10.2.2.2 LZC by Monotonic String Conversion

The second logarithmic LZC construction is more generic; in particular it offers solutions to the fanout problem. Here the idea consists in first converting the input X into a monotonic string Y of the same size, with the same leading 1 as X but only ones to the left of this leading one. Mathematically speaking, $y_i = \bigvee_{j \geq i} x_j$; this is a prefix-OR computation. Efficient hardware implementations of parallel prefix computations have already been surveyed in Sect. 5.3.6. For instance, a Kogge-Stone structure [KS73] will compute the parallel prefix in logarithmic time with a fanout limited to two (see Fig. 5.19). The result is a kind of a thermometer or unary code, where n bits in a row are 1 to encode number n .

By ANDing this vector Y with itself negated and shifted by one bit position, one gets a bit vector $Z = s_{w_X-1} \dots s_1 s_0$ consisting of only zeros, except at the position of the leading one. This is a one-hot code. From this last vector, each bit of the count C can be computed as an $w_X/2$ -wide OR as follows:

- c_0 is the parity of the count and can be obtained by OR-ing every other bit of Z .
- c_{k-1} will be one iff the 1 in Z belongs to its lower half: it can be obtained by OR-ing all the bits of this lower half.
- The intermediate bit c_k can be similarly obtained by OR-ing $w_X/2$ bits grouped in consecutive chunks of k bits.

Figure 10.12 sketches the corresponding architecture and applies it to an example.

This approach is easily adapted to LOC or LBC. It is also well suited to *leading-zero anticipation* techniques, which allow to reduce the delay of floating-point adders and fused multiply-add by computing, in parallel to an addition, an approximation of the LZC of the result of this addition [Ng93; Ino94; SN01]. Note that there exists alternative thermometer code to binary converters which are based on compressor trees [Bay16] that could be applied to the result of the parallel prefix OR.

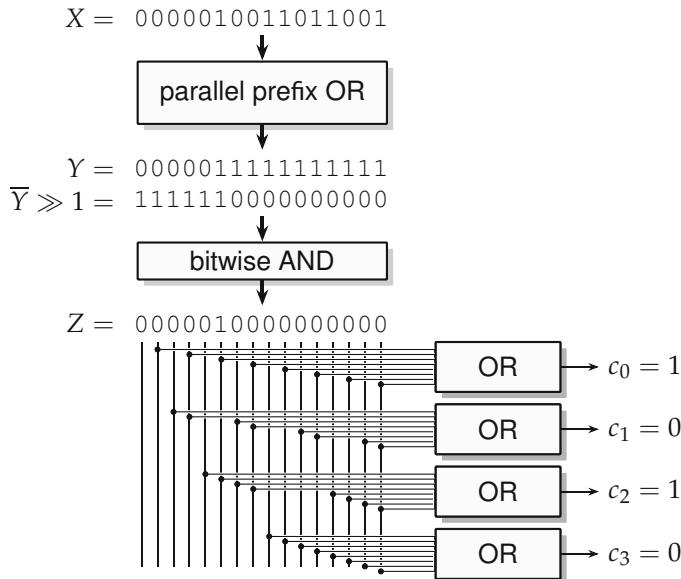


Fig. 10.12 LZC computation by monotonic string conversion.

10.3 Normalizer (Combined Lzc + Shifter)

As the name suggests, the typical use of a normalizer is to bring a floating-point result in normal form (see Sect. 2.5 and Chap. 11). As illustrated in Fig. 10.13, the normalizer inputs a word X , counts its leading zeroes, and shifts it to the left to remove these zeroes, so that its leading one is placed at its MSB. It outputs the shift distance C (which is also the count of leading zeroes in X) along with the shifted result R .

When used for the standard sign/magnitude floating-point formats, a normalizer counts the leading zeroes. Other formats where the significand is in two's complement (such as Posit [GY17]) require normalizers that count the leading sign bit. More generally, a normalizer can be used to convert a fixed-point number into the normal form of a floating-point format. In this case, R becomes the significand of the floating-point output, and C becomes its exponent (plus or minus a constant).

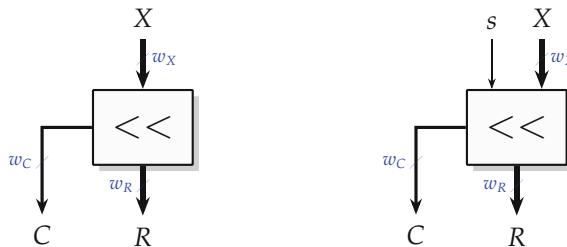


Fig. 10.13 Normalizer counting zeroes (left) or counting the sign bit input as s .

The generic interface to a normalizer generator is given in Fig. 10.14.

Here, it makes no sense to have $w_R > w_X$, since a normalizer essentially removes information from X to encode it more compactly in C . The case when $w_R < w_X$ happens, for instance, in the conversion of a wide fixed-point format into a more compact floating-point one. In such case, it may be useful to also compute the OR of the rightmost bits that won't make it in R and output it as a sticky bit: this information is needed to ensure the “ties to even” rule (see Sect. 3.1.3). An example will be given in Chap. 21. Within floating-point operators, normalizer typically require $w_R = w_X$.

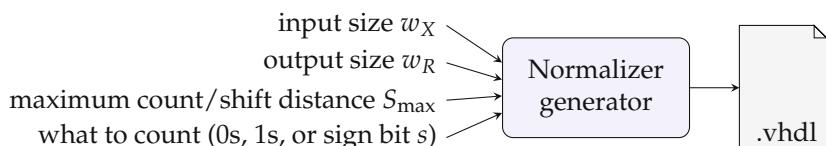


Fig. 10.14 Interface to a generic normalizer generator.

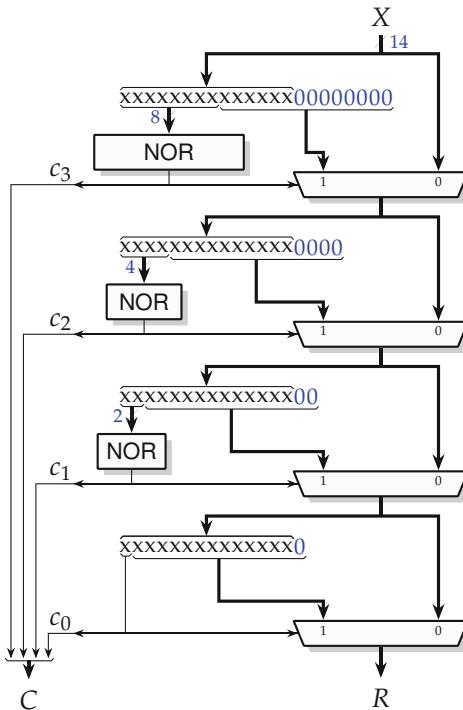


Fig. 10.15 Architecture of a 14-bit combined leading zero counter and left shifter.

Hands on: Normalizer in FloPoCo

The following command produces the normalizer of the above figure:

```
flopoco Normalizer wX=14 wR=14 maxShift=14
```

Optional arguments are `countType` and `computeSticky`.

Note that such a fused normalizer is not relevant when building an IEEE floating-point adder or Fused Multiply-Add (FMA): in such cases there is some arithmetic to perform on the count to ensure proper support of subnormal numbers [Mul+18]. However, it can be used for application-specific floating-point formats without subnormals, such as the FloPoCo Nfloat format of Sect. 2.5.3, for posit arithmetic [GY17; UFD19], and in the internal construction of some elementary functions [DD07b; DD07a].

The simplest implementation of a normalizer is a while loop, shifting left by one bit while the MSB is 0. When implemented in hardware, this solution is very cheap but has a variable latency and cannot be pipelined.

A pipelinable constant-latency solution is the use of an LZC on X , followed by a shifter whose S input is connected to the C output of the LZC. As we have seen, both operations can be performed in time logarithmic in w_X .

However, the two logarithmic-time approaches of Sect. 10.2.2 produce all their count bits in parallel, so there is no way to overlap the computation of the LZC with the shift.

Conversely, the naive LZC of Sect. 10.2.1 can be very naturally merged with a barrel shifter. Since the bits of C arrive MSB first, it is natural here to use the same order for the shift levels. The resulting architecture is shown in Fig. 10.15. Strictly speaking, for the common case $w_X = w_R = S_{\max}$, its area grows as $\mathcal{O}(w_X \log_2 w_X)$, and its delay as $\mathcal{O}((\log_2 w_X)^2)$, but again the practical delay is closer to $\log_2 w_X$ in FPGAs thanks to the fast carry logic based NOR computation.

10.4 To Read Further

The shifters and leading zero counters studied in this chapter are essentially built out of multiplexers. In FPGAs, these multiplexers will be implemented in the programmable logic, consuming logic resources. However, the programmable routing architecture of most FPGAs is also based on huge numbers of multiplexers, controlled by configuration bits. As shifters are ubiquitous in floating-point based application-specific arithmetic, it has been proposed [Moc+12] that some of these routing multiplexers should also be configurable as run-time multiplexers, so that they can be used to implement shifters and LZCs.

In some applications, only a set of discrete shifts are necessary, in contrast to a continuous shift range. Those highly application-specific shifters are called sporadic shifters [Che+18], and it is possible to derive an optimized organization of the multiplexers from the set of shift values [Che+18].

Since a left shift is a multiplication by a power of two, it is possible to use a multiplier to implement a left shift. This is why some AMD cores for floating-point addition may consume DSP blocks [Xil19]. The shift value must first be one-hot encoded (using FPGA logic), but this is much faster and cheaper than a complete shifter.

When the input X is constant, it becomes possible to tabulate all the possible shifted values. In [LP13], the shift following a polynomial evaluation is replaced by a shift of the polynomial coefficients, tabulated in a block RAM addressed by the shift value.

References

- [Bay16] Eugen Bayer. "Entwurf und Systemintegration eines Signalsyntheseverfahrens auf programmierbaren Logikbausteinen". PhD thesis. University of Kassel, 2016 (cit. on p. [322](#)).
- [Che+18] Jiajia Chen, Chip-Hong Chang, Yujia Wang, Juan Zhao, and Susanto Rahardja. "New Hardware and Power Efficient Sporadic Logarithmic Shifters for DSP Applications". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.4 (2018), pp. 896–900 (cit. on pp. [313](#), [325](#)).
- [DD07a] Jérémie Detrey and Florent de Dinechin. "Floating-Point Trigonometric Functions for FPGAs". In: *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2007, pp. 29–34 (cit. on pp. [212](#), [324](#)).
- [DD07b] Jérémie Detrey and Florent de Dinechin. "Parameterized floating-point logarithm and exponential functions for FPGAs". In: *Microprocessors and Microsystems, Special Issue on FPGA-based Reconfigurable Computing* 31.8 (2007), pp. 537–545 (cit. on p. [324](#)).
- [DI15] Florent de Dinechin and Matei Iștoan. "Hardware implementations of fixed-point Atan2". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2015, pp. 34–41 (cit. on pp. [307](#), [316](#), [761](#)).
- [Din+10] Florent de Dinechin, Mioara Joldes, Bogdan Pasca, and Guillaume Revy. "Multiplicative square root algorithms for FPGAs". In: *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2010, pp. 574–577 (cit. on pp. [13](#), [303](#), [307](#), [356](#)).
- [GKP94] Ronald L. Graham, Donald Ervin Knuth, and Oren Patashnik. *Concrete Mathematics - A Foundation for Computer Science*. Addison-Wesley Professional, 1994 (cit. on p. [320](#)).
- [GY17] John L. Gustafson and Isaac T. Yonemoto. "Beating Floating Point at its Own Game: Posit Arithmetic". In: *Supercomputing Frontiers and Innovations* 4.2 (2017), pp. 71–86 (cit. on pp. [323](#), [324](#)).
- [Ino94] Genichiro Inoue. "Leading one anticipator and floating point addition/subtraction apparatus". U.S. pat. 5343413. 1994 (cit. on p. [322](#)).
- [KS73] Peter M. Kogge and Harold S. Stone. "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations". In: *IEEE Transactions on Computers* C-22.8 (1973), pp. 786–793 (cit. on p. [321](#)).
- [LP13] Martin Langhammer and Bogdan Pasca. "Elementary Function Implementation With Optimized Sub-Range Polynomial Evaluation". In: *International Symposium on Field-Programmable Cus*

- tom Computing Machines (FCCM)*. IEEE. 2013, pp. 202–205 (cit. on p. 325).
- [Moc+12] Yehdhih Ould Mohammed Moctar, Nithin George, Hadi Parandeh-Afshar, Paolo Ienne, Guy G.F. Lemieux, and Philip Brisk. “Reducing the Cost of Floating-Point Mantissa Alignment and Normalization in FPGAs”. In: *Field Programmable Gate Arrays*. ACM, 2012, pp. 255–264 (cit. on p. 325).
- [Mul+18] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldeş, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. 2nd ed. Birkhäuser Boston, 2018 (cit. on pp. 11, 24, 311, 314, 324, 330, 332, 336, 346, 347, 349).
- [Ng93] Kenneth Ng. “Method and apparatus for exact leading zero prediction for a floating-point adder”. U.S. pat. 5204825. 1993 (cit. on p. 322).
- [Okl94] Vojin G. Oklobdzija. “An Algorithmic and Novel Design of a Leading ZeroDetector Circuit: Comparison with Logic Synthesis”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 2.1 (1994) (cit. on p. 320).
- [SN01] Martin M. Schmookler and Kevin J. Nowka. “Leading Zero Anticipation and Detection - A comparison of methods”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2001, pp. 7–12 (cit. on pp. 322, 342).
- [UFD19] Yohann Uguen, Luc Forget, and Florent de Dinechin. “Evaluating the hardware cost of the posit number system”. In: *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2019 (cit. on pp. 316, 324).
- [Xil19] *Floating-Point Operator v7.1, LogiCORE IP Product Guide PG060*. Xilinx. 2019 (cit. on p. 325).



CHAPTER 11

Basic Floating-Point Operators

*It makes me nervous to fly on airplanes
since I know they are designed using floating-point arithmetic.*

Alston Householder

Relax. Today's planes are piloted using floating-point arithmetic.

The authors

This chapter shows how to build the operators for the basic operations (addition and subtraction, multiplication, division, and square root) in floating point. Specialized floating-point operators (such as squarers and constant multipliers) and fused floating-point operators (such as fused multiply-add, combined sum and difference, or sum of squares) will be reviewed in Chap. 15. For each operation, we start with the construction of simple but non-standard operators suitable for hidden application-specific datapaths. Then, refinements for improved standard compliance or improved performance are presented.

Operations on floating-point numbers are naturally expressed in terms of fixed-point operations on the significand and exponent. For instance, for a plain format, the product of two floating-point numbers is obtained by multiplying the significands and adding the exponents. Floating-point addition is built out of fixed-point addition, shifters, and leading-zero counters. The hardware cost of each of these subcomponents scales well (between linearly and quadratically) with the precision. This simplicity is actually the main advantage of floating-point arithmetic over alternatives such as the Logarithm Number System (LNS) format (see Sect. 2.6). Although the latter has a comparable dynamic range and, arguably, better number representation properties, the implementation of LNS addition is complex, and its cost scales poorly to large precision.

The construction of floating-point operators may become tricky for three main reasons:

1. The correct rounding of the result is sometimes quite expensive, in particular compared to faithful rounding.
2. The support of subnormal numbers adds to the complexity and cost of the operators.
3. The literature offers many tricks to enhance the performance (in particular the latency) of the operators, usually at a cost.

These three issues will be briefly surveyed in the context of application-specific arithmetic.

In particular, the support of correct rounding and subnormals should be carefully assessed for operators belonging to application-specific floating-point datapaths. Indeed, for a given application-level accuracy, it is often cheaper to use faithful operators than correctly rounded ones. Proper subnormal handling is also often more expensive than adding one bit to the exponent size, which is an option in embedded application-specific circuits. Of course, the format itself can (and should) be sized to the application, as discussed in Sect. 2.5.

Choosing the proper level of compliance to IEEE 754 [754-19] is very context-specific. As a general rule, as soon as an architecture is programmable and offers standard formats, users will expect IEEE 754 compliance, and the predictability and reproducibility offered by standard compliance are worth its overhead. This is perfectly illustrated by the history of floating-point support in graphics processing units (GPUs): early architectures took liberties with the IEEE standard in the interest of performance, and it was acceptable as long as programming GPUs was a niche activity. However, as soon as GPU programming went mainstream, full IEEE compliance became the norm. Conversely, for an invisible operator buried in the depth of an application-specific datapath, it may make perfect sense to use non-standard formats and relax compliance in the interest of efficiency [EL11].

In this spirit, this book focuses on rounding to the nearest or faithful rounding. The reader should be aware that we also leave aside several important features of the IEEE 754 standard, such as exception handling¹ and the directed rounding modes (toward zero, $+\infty$, or $-\infty$). For the construction of standard IEEE 754 operators, the interested reader is referred to the Handbook of Floating-Point Arithmetic [Mul+18].

¹ The IEEE 754 standard [754-19] defines five exceptions (*Invalid*, *Overflow*, *Underflow*, *DivideByZero*, and *Inexact*) that can be trapped by software to manage the respective situations. Software may also ignore these exceptions, because the hardware returns a value in each of these situations (a NaN for *Invalid*, an infinity for *Overflow* and *DivideByZero*, a subnormal result for *Underflow*). In this book we assume that application-specific hardware will do without raising these exceptions. Our reader having a need for any of them should be aware that they have been well thought out in the IEEE 754 standard.

The remaining sections of this chapter deal, respectively, with addition and subtraction, multiplication, division, and square root. Note that this usual order is almost by decreasing complexity: floating point management is the most complex for addition and the simplest for square root.

11.1 Floating-Point Addition and Subtraction

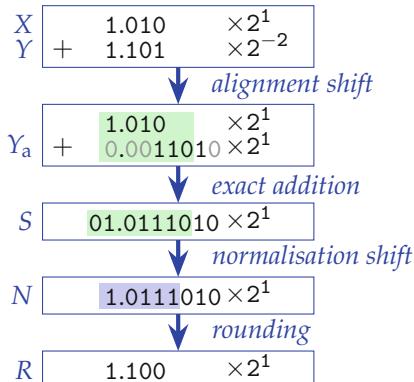
Let us first point out that subtraction can be reduced to addition by exploiting the identity $X - Y = X + (-Y)$, valid for all floating-point numbers except Not a Number (NaN)—the latter being treated equivalently in addition and subtraction. Hence, any floating-point adder can be converted into an adder/subtractor at the cost of a single XOR gate to negate the sign bit of the second operand. For this reason, we focus on addition in the following, including the addition of numbers of different signs.

11.1.1 General Considerations and Terminology

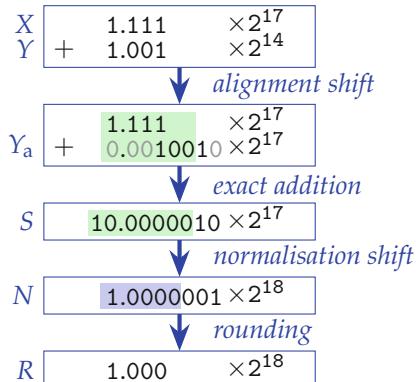
Floating-point addition is complex to implement because of the large number of possible situations. Let us first define the general concepts with the help of examples, in Figs. 11.1, 11.2 and 11.3, using a toy binary floating-point format with 3 bits of fraction (plus an implicit leading one). The reader unfamiliar with binary arithmetic is invited to redraw these examples in decimal: binary is very similar to decimal here. Also note that we keep the exponent in decimal for clarity: its encoding is of little significance to understand the algorithms involved.

Table 11.1 Special cases of floating-point addition for round to nearest when adding operand X (columns) with operand Y (rows).

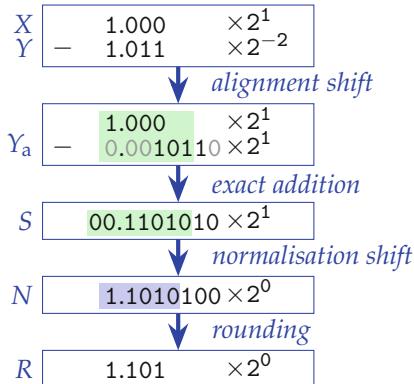
+ \ (Sub)normal Y	NaN	-0	+0	-∞	+∞	(Sub)normal X
NaN	NaN	NaN	NaN	NaN	NaN	NaN
-0	NaN	-0	+0	-∞	+∞	X
+0	NaN	+0	+0	-∞	+∞	X
-∞	NaN	-∞	-∞	-∞	NaN	-∞
+∞	NaN	+∞	+∞	NaN	+∞	+∞
(Sub)normal	NaN	Y	Y	-∞	+∞	$ X + Y $



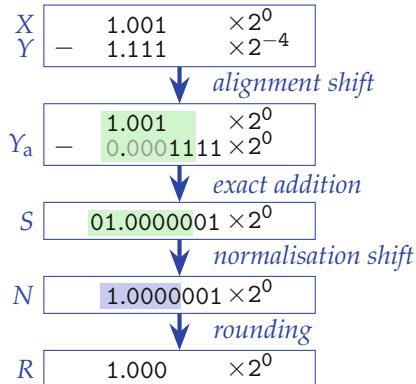
(a) typical "far case" addition



(b) "far case" addition with 1-bit right normalization



(c) "far case" subtraction with 1-bit left normalization



(d) worst-case alignment shift for "far case" addition

Fig. 11.1 "Far case" examples of floating-point additions.

First, there are many exceptional and trivial cases which are summarized in Table 11.1. Note that the addition of two infinities of the same sign returns this infinity, whereas the addition of two infinities of different signs returns a NaN. A choice must be made for the addition of signed zeroes of different signs, mostly to ensure reproducibility among computing systems. The choices in Table 11.1 are taken from the IEEE 754-2019 standard [754-19] for the round to nearest mode. The reader interested in the other rounding modes is referred to the standard or to [Mul+18].

We are not completely done with exceptional case handling: infinities and zeroes may still appear as the result of operations on normal numbers (overflows and underflows).

Once these input exceptional cases have been ruled out, the inputs are possibly swapped to ensure that the exponent of X is larger than (or equal to) the exponent of Y .

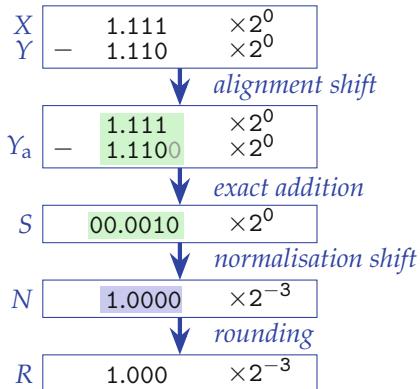
$$\begin{array}{r}
 1.001 \times 2^0 - 1.111 \times 2^{-5} \\
 - \\
 = 1.00010001 \times 2^0 \\
 \text{rounded to } 1.001 \times 2^0
 \end{array}$$

Fig. 11.2 Example of floating-point addition with one operand rounded off.

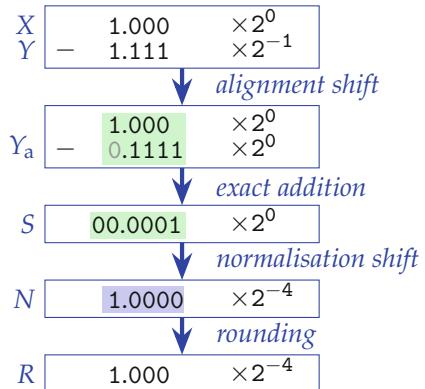
There is one more common trivial situation, when the exponent difference is so large that the smaller addend is strictly smaller than the half-ulp of the larger. Then, the sum is, by definition, equal to the input of largest magnitude. This happens as soon as the exponent difference is strictly larger than $w_F + 1$, as illustrated in Fig. 11.2. Therefore, the exponent difference is computed early in the datapath and usually also used to decide the swap, using a variation on the **exponent difference and swap** architecture depicted in Fig. 11.4. A compound adder (see Sect. 5.3.7) may be used here, but as exponents are small numbers (up to 15 bits for quad precision), the overhead of a fast adder may not be justified.

Now, we have an actual addition or subtraction to perform when the exponent difference is at most $w_F + 1$. Then, floating-point addition consists of four steps illustrated in Figs. 11.1 and 11.3:

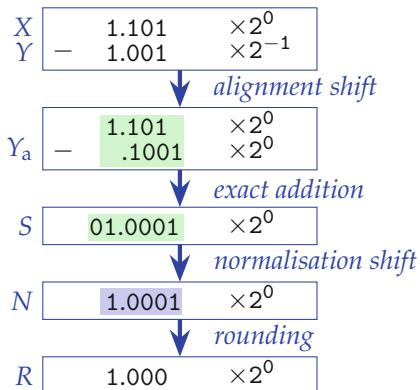
1. In the **alignment shift** step, the representation of Y is modified so that its exponent becomes equal to that of X . In practice, the significand of Y is shifted right by a number of positions equal to the exponent difference $E_X - E_Y$, which is at most $w_F + 1$ as already observed. Once both inputs are aligned, the exponent of the result is tentatively set to that of X .
2. The **exact addition** step then adds the two significands. After this step we have an exact representation of the sum, but it is not always normalized (i.e., with exactly one non-zero bit to the left of the point).
3. The **normalization shift** step brings the leading non-zero bit of the sum to the left of the point and updates the result exponent accordingly. It will be useful in the sequel to distinguish two cases of normalization.
 - If the exponents are different enough (“far” case, illustrated in Fig. 11.1), then the exponent of the result will be that of X (Fig. 11.1a), plus or minus one (Fig. 11.1b–d).



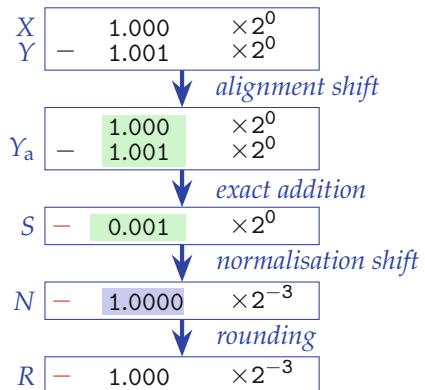
(a) example of cancellation when the exponents are the same



(b) example of cancellation with an exponent difference of one



(c) example of near case subtraction requiring a rounding



(d) example of near case subtraction changing the sign of the result

Fig. 11.3 “Near case” examples of effective subtractions.

- If the exponent difference is 0 or 1, and if the operation is an effective subtraction,² then a *cancellation* may occur. This means that a large normalization shift may be needed. This “near” case is illustrated in Fig. 11.3. Note that in this case, by definition, the alignment shift is small (0 or 1 bit only).

² It may be the subtraction of two numbers with the same sign or the addition of numbers with different signs.

4. Finally, the **rounding step** rounds the normalized significand to the target format. In very rare cases, this step may again entail an exponent update (e.g., when rounding 1.1111110×10^0 to 1.000×10^1). These cases are simple to handle thanks to the choices made for the encoding of the subnormals and the exponent: the round bit is added to the $(w_E + w_F)$ -wide bit vector formed by the concatenation of biased exponent and fraction. Carry propagation from the significand to the exponent then properly updates the exponent while resetting the significand fraction bits in these cases. See Table 2.1 for an illustration.

In Figs. 11.1 and 11.3, the bit vectors are represented with a size that captures the worst-case situations: a shift of at most $w_F + 1$ bits for the shifted significand and at most an overflow bit after the addition. However, there is one more important observation to be made. In the “far” case, the rightmost shifted bits (line Y_a of Fig. 11.1) are never used. The actual addition/subtraction needs only to be computed on bits up to the rounding position, which may be at positions (with respect to the significand of X) $-w_F - 1$ (Fig. 11.1a and d), $-w_F$ (Fig. 11.1b), or $-w_F - 2$ (Fig. 11.1c). The actual addition/subtraction to perform needs only to provide these bits: it inputs two $(w_F + 3)$ -bit numbers and outputs a $(w_F + 4)$ -bit result (these bits are highlighted in green in Figs. 11.1 and 11.3).

The rightmost $w_F - 1$ bits of the shifted significand Y_a are traditionally compressed in a *sticky* bit whose value is 0 if and only if all these rightmost bits are 0. With round to nearest, this information is used to decide the (quite rare) tie cases, i.e., the cases when the normalized exact sum N is exactly between two floating-point numbers. With our toy $w_F = 3$ system, this would happen, e.g., for $N = 1.101\ 1000$ or $N = 1.110\ 1000$. The main IEEE 754 rounding mode, “rounding to nearest, ties to even,” mandates that in such cases the significand to be returned should be the one ending with a 0 (in the previous examples, $R = 1.110$ in both cases). This way the ties are sometimes rounded up, sometimes down, and they should not entail any statistical bias.

The sticky bit computation is relatively cheap to implement (see Sect. 10.1.5), but it should be clear that in an application-specific context, it can be replaced with “round to nearest, ties to away,” which simply adds a 1 at position $-w_F - 1$ of the mantissa of N without consideration to any of the bits to the right.

Origin of the “ties to even” rule

The sticky bit computation is necessary to implement correct rounding in *directed* rounding modes. For instance, $N = 1.101\ 0001$ should be rounded up to 1.110, but $N = 1.101\ 0000$ should be rounded up to 1.101. As IEEE-compliant hardware must support these directed rounding modes, it must include hardware that computes the sticky bit. The ties-to-even rule for round to nearest was chosen as a good use of this hardware when rounding to nearest.

In the “near” case (Fig. 11.3), the situation is slightly simpler: the sticky bit is always 0 and no overflow may occur. Altogether the effective subtraction can be one bit smaller. Any cancellation situation entails that the result is exact (this is related to Sterbenz Lemma [Mul+18]). However, there are still situations that require rounding (Fig. 11.3c).

In the “far” cases, the sign of the result is always that of X , whether the operation is an effective addition or an effective subtraction. However, in the “near” case when the exponent difference is 0, the result may become negative (Fig. 11.3d). As the final floating-point result is expected to have a positive significand, we need to compute the opposite of the negative significand. This is another case for a compound adder computing $|X - Y_a|$ and also outputting the “sign change” bit (in red in Fig. 11.3d).

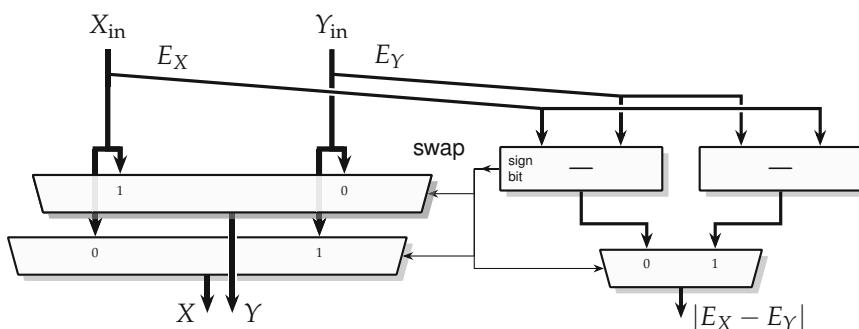


Fig. 11.4 Exponent difference and swap.

11.1.2 Baseline Addition Architecture

With all these considerations, a baseline architecture of minimal cost that manages all these cases for the Nfloat format or IEEEfloat format without subnormals is given in Fig. 11.5. It implements the round to nearest, ties to away rounding rule.

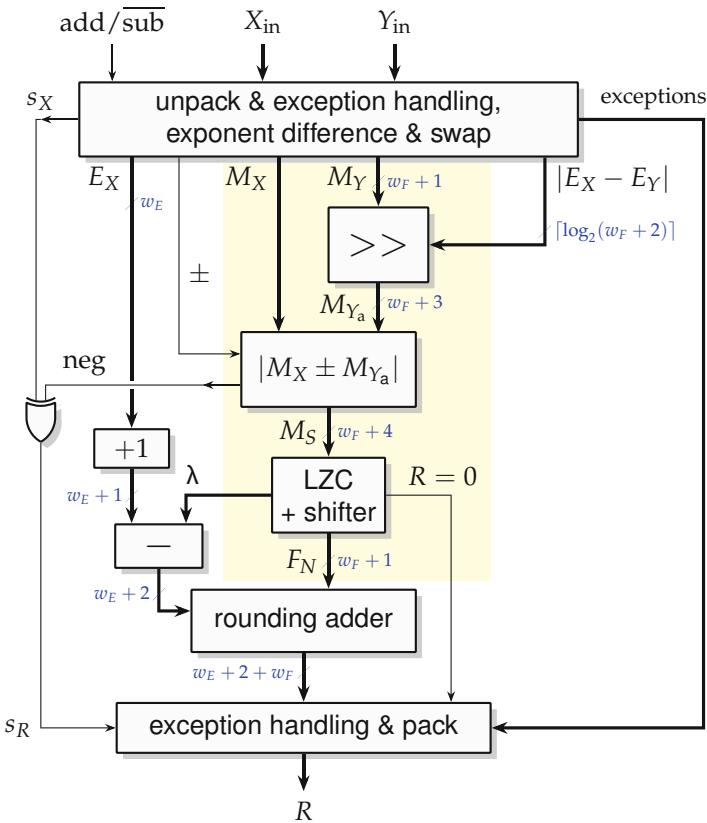


Fig. 11.5 Baseline floating-point adder.

The purpose of this drawing is to show the main processing blocks and datapath widths. Many details are missing; some are listed below, and many open-source cores are available for the interested reader.

- This operator performs an addition or a subtraction depending on the input add/sub. Sign management is generally trivial and not detailed. The signal labeled \pm defines whether the operation is an effective addition or an effective subtraction of the significands.
- For Nfloat, “unpack & exception handling” simply means combining possible exceptional cases as per Table 11.1. For IEEEfloat, this block must first decode infinities, NaN and subnormals encoded as special exponent values. In a baseline architecture, subnormals are converted to zeroes – proper subnormal support is detailed below in Sect. 11.1.3.
- The exponent difference must be saturated to $w_F + 2$ when it exceeds $w_F + 2$. This way, the significand M_Y is completely shifted out, and the $|M_X \pm M_{Y_a}|$ block simply adds 0 in the corresponding cases.

- Similarly, the leading zero count has to be saturated: if it equals $w_F + 4$, it means that a full cancellation occurred, and the result is zero: this case will be managed by the [exception handling & pack] block.
- The best implementation of the $[M_X \pm M_{Y_a}]$ component is technology-dependent. It may be similar to the right part of Fig. 11.4, or it may use a variation on the compound adder of Sect. 5.3.7. In any case this component also outputs the sign bit of $M_X \pm M_{Y_a}$ (the signal labeled neg) which is used to determine the sign bit of the result.
- In the FloPoCo implementations, the swap decision is taken on a comparison on the full numbers (exponent concatenated with fraction). This ensures that $M_X > M_Y$ (even in the near cases), hence that the sign of the result is that of X. Thus, the $[M_X \pm M_{Y_a}]$ block becomes simply $[M_X \pm M_{Y_a}]$. This is a sensible choice on FPGAs because this early large comparison is fast and cheap thanks to fast carry logic (see Sect. 6.3, p. 148).
- The [LZC/shifter] component normalizes the sum significand M_S and then discards the leading 1 to output only the fraction F_N of the normalized significand (extended with a rounding bit at the LSB). It also outputs the leading zero count λ .
- The exponent update subtracter must not subtract λ , but $\lambda - 1$. Indeed, leading zeroes are counted on the bit vector labeled S in Fig. 11.1, and this -1 accounts for the possible 1-bit right shift in Fig. 11.1b. To hide the delay of this extra addition, the tentative exponent E_X is instead incremented before the subtraction of λ .
- The exponent update may entail either an overflow (in the situation of Fig. 11.1b when E_X was already the maximum possible value) or an underflow (when the updated biased exponent becomes negative). To distinguish between these cases in the [exception handling & pack] block, the output of the exponent update subtractor is a $(w_E + 2)$ -bit number, with one sign bit and one positive overflow bit on top of the w_E exponent bits.
- The rounding adder simply adds a 1 at the LSB of the concatenation of the $(w_E + 2)$ -bit updated exponent and the $(w_F + 1)$ -bit normalized fraction (from the line labeled N in Figs. 11.1 and 11.3). It then discards this LSB bit, transmitting the final w_F -bit fraction and the (possibly updated again) $(w_E + 2)$ -bit exponent to the [exception handling & pack] block.
- The latter merges the exception information from the initial exception handling with that encoded in the exponent. Its work depends whether an Nfloat or an IEEEfloat format is used as the output format (in particular there are more overflow/underflow situations in IEEEfloat to make room for the encoding of 0, ∞ , and NaN in the exponent field). It also concatenates the sign of the result.

As a final remark, note that a biased exponent has no overhead here, as it impacts neither exponent difference nor exponent update.

There are two ways to improve this baseline adder.

- **Functionally**, although the baseline adder will often be perfectly adequate when completely hidden in an application-specific datapath, some applications mandate IEEE 754 compliance. Sect. 11.1.3 discusses the main corresponding changes: support of round to nearest with ties to even rule and subnormal support.
- In terms of **performance**, there are several standard techniques that can reduce the latency of the floating-point adder at the expense of more hardware. They are briefly reviewed in Sect. 11.1.4.

To our knowledge, degrading the accuracy of a floating-point adder from round to nearest to faithful accuracy would bring little benefit.

11.1.3 From Baseline Adder to IEEE 754 Compliance

The first missing feature is the support of the ties-to-even rule. This requires to compute the OR of all the bits shifted to the right of the rounding bit. The overhead is either to replace the first right shifter (which is a quasi-barrel shifter in the baseline adder) with a full shifter, followed by a wide OR, or to integrate the sticky computation in the shifter as detailed in Sect. 10.1.5. This first sticky bit is then possibly updated after the normalization shift (in the case of Fig. 11.1b). A tie case is then detected when the round bit (the LSB of F_N) is 1 and the sticky is zero. In this case the fraction should be increased if the next-to-LSB bit of F_N is a 1, so that it becomes a 0 and the final fraction becomes even. This adds a handful of gates to the **rounding adder** block: the bulk of the overhead is in the sticky computation.

Hands on: Baseline floating-point addition

At the time of writing this book, there is no baseline adder (rounding to the nearest with ties up) in FloPoCo. The following command implements a baseline adder, rounding to the nearest with ties to even, for the Nfloat(8,23) format.

```
flopoco FPAdd we=8 wf=23
```

The second missing feature is subnormal support, and we must distinguish two issues:

- The proper management of input subnormals is relatively simple and can be hidden in the first unpack box. First, two “is normal” bits n_X and n_Y are determined out of the exponent fields of X and Y (see Sect. 2.5.1, p. 48). Following (2.18), these bits are used twice: (1) they are appended to the fraction ($M_X = n_X.F_X$ and $M_Y = n_Y.F_Y$, where the baseline adder systematically appended a 1 bit), and (2) the exponent difference becomes $E_X - n_X - (E_Y - n_Y)$. This is enough to ensure proper alignment in the event of subnormal input(s).
- The leading zero count must be saturated so that $E_X - \lambda$ does not become negative. This requires a comparison and a multiplexer between the leading zero counter and the shifter. It also prevents the use of a combined leading zero count and shifter.

Otherwise subnormals need no special management in the rounding adder, thanks to the clever encoding of Table 2.1.

Altogether, the area overhead of subnormal support is limited, since it mostly concerns the exponent datapath, but it impacts the critical path.

Hands on: IEEE-compliant floating-point adders

The reader interested in the full details of an IEEE-compliant floating-point adder may obtain one as follows:

```
flopoco IEEEFPAdd wE=8 wF=23
```

Many other open-source sources exist, in particular in more modern hardware development frameworks such as Chisel. For a parametric IEEE adder in synthesizable C++, look up the Marto (Modern ARithmetic Tools for high-level synthesis) project.

11.1.4 Dual-Path Architectures and Other Speculative Techniques

The critical path of our adder (baseline or IEEE-compliant) includes several blocks which have a latency in $\log_2 w_F$: the alignment shifter, the significand adder, the LZC, the normalization shifter, and the rounding adder. To reduce it, a crucial observation is that the “far” and “near” cases are exclusive: when a large alignment shift is needed (“far” cases), there is no need for a large LZC and normalization shifter. Conversely, when the latter are used, there was no large alignment shift (“near” cases).

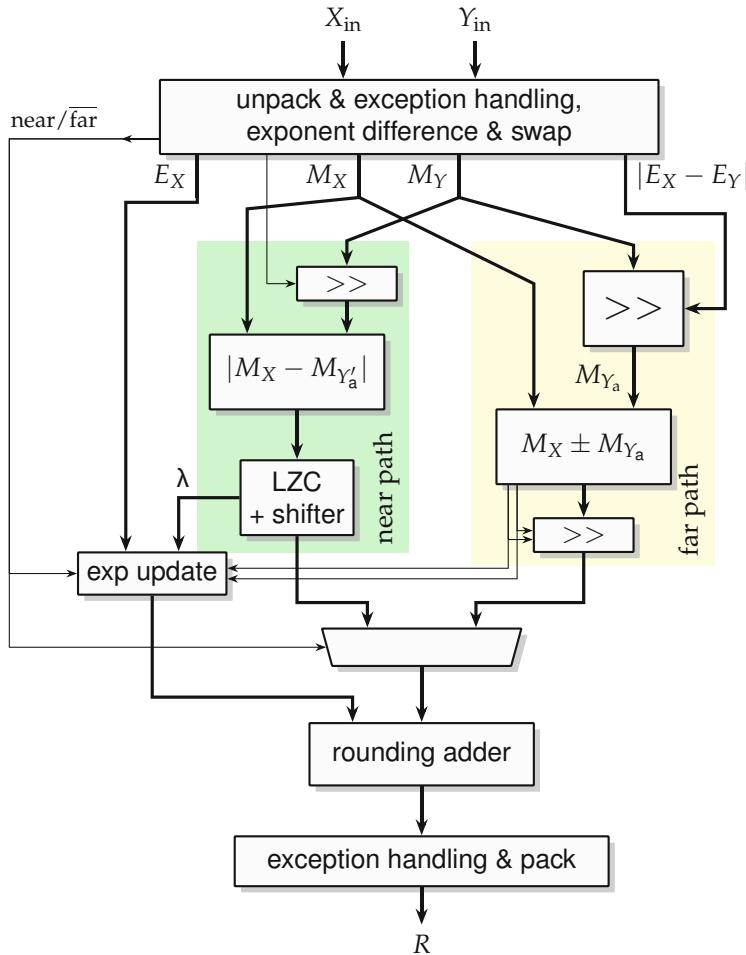


Fig. 11.6 Overview of a dual-path floating-point adder.

In *dual-path* architectures, the three blocks highlighted in Fig. 11.5 are therefore replaced with two parallel paths:

- The “far” path manages the “far” cases. It is composed of a large alignment shifter, an adder/subtractor, and a very small (2-bit) LZC/normalization shifter.
- The “near” path manages the “near” cases. It is composed of a very small (1-bit) alignment shifter, followed by a compound adder computing $|M_X - M_{Y_a}|$, then a large LZC, and a normalization shifter.

A multiplexer selects the output of the relevant path based on the exponent difference. Altogether, the overhead of this approach is limited to this mul-

triplexer and a duplication of the significand adder. Remark that no sticky computation is needed on the near path.

Sign management in the near path is similar to the baseline architecture, therefore not detailed in Fig. 11.6. The sign of a far-path result is that of X.

Hands on: Dual-path baseline floating-point addition

The following command implements a dual-path floating-point adder for the Nfloat(8,23) format.

```
flopoco FPAdd we=8 wf=23 dualpath=true
```

Without subnormal support, the two paths are quite well balanced in terms of critical path. However, subnormal support entails the use of separate LZC and normalization shifter: the near path then includes three slow ($\log_2 w_F$) operations versus two for the far path.

A classic solution to this problem is *leading zero anticipation*, where the LZC is replaced by a different component, inputting M_X and M_{Y_a} and running in parallel to their subtraction. It computes an approximation of the leading zero count λ that may be wrong by one bit and must be corrected by a final multiplexer. The reference on leading zero anticipation techniques is a survey by Schmookler and Nowka [SN01] and recent improvements by Lutz [Lut17].

To conclude this overview, let us mention an extreme approach that allows drastic latency reductions at the expense of drastically higher area: speculation. D. Lutz mentions [Lut19] that 1-cycle double-precision addition at 1 GHz is possible by the following:

- Computing all four possible near-path differences: $M_X - M_Y$, $M_Y - M_X$, $M_X - (M_Y \gg 1)$, $M_Y - (M_X \gg 1)$ in parallel, all this while the exponents are being subtracted.
- Also while the exponents are being subtracted, starting the far path shift on both operands M_X and M_Y .
- Computing, as soon as M_X is decided and in parallel to the end of the shift alignment, the two possible additions of rounding bits to M_X : in case of effective addition, $M_X + 0.5$ and $M_X + 1$; in case of effective subtraction, $M_X + 0.5$ and $M_X + 0.25$. This anticipates all the possible rounding cases in Fig. 11.1.
- Finally, computing three far-path additions in parallel, in order to speculatively add to the aligned M_{Y_a} either M_X (unrounded case) or the result of the two previous additions (two rounded cases).

This floating-point adder includes two 54-bit incrementers and seven 54-bit adders in all, along with a large numbers of multiplexers.

Another good illustration of the state of the art in floating-point adder design is an article by Sohn et al. [Soh+22].

11.2 Floating-Point Multiplication

Floating-point multiplication is simpler (although typically costlier) than addition. As for addition, we first present in Sect. 11.2.1 a baseline version that implements round to nearest, ties to away for normalized numbers, and then we enhance it to full compliance (in particular subnormal support) in Sect. 11.2.2. Conversely, a degraded-accuracy (faithful) multiplier is described in Sect. 11.2.3. Finally, Sect. 11.2.4 describes an architectural trick to speed up floating-point multiplication.

11.2.1 Baseline Multiplication Architecture

Beyond the fact that $0 \times \infty$ returns a NaN, exceptional case management (summarized in Table 11.2) is straightforward in multiplication.

Table 11.2 Special cases of floating-point multiplication for round to nearest.

\times	NaN	-0	+0	$-\infty$	$+\infty$	$-\infty < X < 0$	$0 < X < +\infty$
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
-0	NaN	+0	-0	NaN	NaN	+0	-0
+0	NaN	-0	+0	NaN	NaN	-0	+0
$-\infty$	NaN	NaN	NaN	$+\infty$	$-\infty$	$+\infty$	$-\infty$
$+\infty$	NaN	NaN	NaN	$-\infty$	$+\infty$	$-\infty$	$+\infty$
$-\infty < Y < 0$	NaN	+0	-0	$+\infty$	$-\infty$	$[X \times Y]$	$[X \times Y]$
$0 < Y < +\infty$	NaN	-0	+0	$-\infty$	$+\infty$	$[X \times Y]$	$[X \times Y]$

The exact product of two normal numbers $X = (-1)^{s_X} \cdot 2^{E_X - E_0} \cdot M_X$ and $Y = (-1)^{s_Y} \cdot 2^{E_Y - E_0} \cdot M_Y$ is equal to

$$X \times Y = (-1)^{s_X + s_Y} \cdot 2^{E_X + E_Y - 2E_0} \cdot M_X \times M_Y. \quad (11.1)$$

Figure 11.7 shows the architecture of the baseline multiplier that normalizes and rounds this exact product.

Equation (11.1) shows that the exponent of the result is simply the sum of the input exponents. Since the exponent fields are biased, we need to subtract the bias E_0 from their sum.

The significand product $M_X \times M_Y$ is a $(2w_F + 2)$ -bit number that needs to be rounded, but before this it needs to be normalized. Indeed, with $M_X \in [1, 2)$ and $M_Y \in [1, 2)$ (normalized inputs, in the ufix($0, -w_F$) format), we

have $P = M_X \times M_Y \in [1, 4]$ (this is a number in $\text{ufix}(1, -2w_F)$ format). In the case when $P \in [2, 4)$ (product overflow), we need to divide P by 2 (a 1-bit right shift) and add 1 to the exponent. On the significand side, this 1-bit normalization is simply a multiplexer controlled by the leading bit of the product. After this normalization, the only bits of P that need to be kept are those of the target fraction, plus one rounding bit at position $-w_F - 1$. The leading one (now always at position 0) can be discarded. The bits lower than the rounding bit can also be simply discarded: the baseline multiplier thus obtained implements the “rounding to the nearest with ties to away” rule. Note that these lower bits will be needed to manage tie situations for other rounding modes: this will be detailed in Sect. 11.2.2. For our baseline multiplier, however, what comes out of the [1-bit norm] block is a fraction in $\text{ufix}(-1, -w_F - 1)$ format.

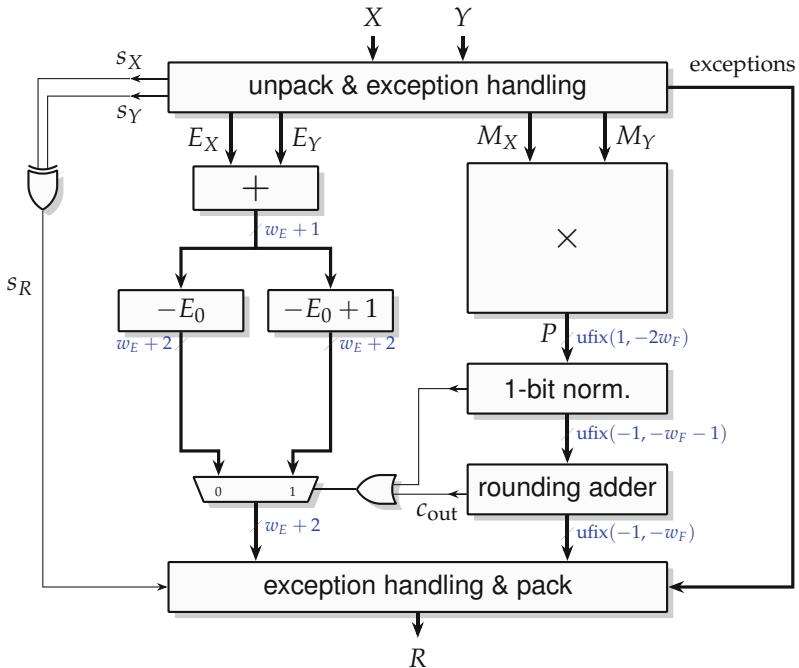


Fig. 11.7 Baseline floating-point multiplier.

On the exponent datapath, there is one subtlety. Two tentative exponent fields are computed: $E_X + E_Y - E_0$, and $E_X + E_Y - E_0 + 1$. The latter is to be used in two cases: when the product overflows, i.e., $P \in [2, 4)$, or when the rounding addition overflows (in which case the fraction becomes all zeroes and the exponent must be incremented). These two situations are actually exclusive: there can be no rounding overflow when there

was a product overflow, since the maximum significand product value is $(2 - 2^{-w_F})^2 = 4 - 4 \cdot 2^{-w_F} + 2^{-2w_F}$, normalized to $2 - 2^{-w_F+1} + 2^{-2w_F-1}$. This makes it possible to use, for the rounding adder, a multiplexer instead of a carry propagation on the exponent bits. This slightly reduces the critical path compared to a normalization adder on $w_E + w_F$ bits as used in the floating-point adder.

As for the adder, the unpack and pack components have more work to do when the inputs are encoded in IEEE-like format. As for the adder, the exponent datapath computes a biased exponent that can overflow or become negative (underflow). Distinguishing these two cases requires keeping the exponent on $w_E + 2$ bits (including a sign and an overflow bit) before the final exception handling.

Hands on: Baseline floating-point multiplication

The following command implements a floating-point multiplier for the Nfloat(8,23) format.

```
flopoco FPMult we=8 wf=23
```

11.2.2 From Baseline Multiplier to IEEE 754 Compliance

Round to nearest with ties to even adds to Fig. 11.7 the computation of the sticky bit on the lower bits of P . This is a small OR tree, and as these bits arrive first, the sticky computation does not add to the critical path.

Subnormal handling, conversely, is quite expensive both in area and latency. The products of two normal numbers can be subnormal, and the product of a normal by a subnormal can become normal; in short we have to manage subnormals both in inputs and outputs. Fortunately the product of two subnormals is always rounded to 0; therefore on the input, we only have to manage the case when one of the inputs is subnormal.

Conceptually, the simplest technique is to normalize the subnormal input first (using an LZC and a shifter and a 1-bit wider exponent), then use the multiplier in Fig. 11.7, and then possibly denormalize the result if it is subnormal. This more than doubles the latency compared to Fig. 11.7.

A better idea is to perform only one shift. Indeed, the true exponent of the result can be deduced from E_X, E_Y and possibly the leading zero count λ on the subnormal input. Besides, the significand multiplier will handle as well inputs whose implicit bit is the “is normal” bit (see Sect. 2.5.1, p. 48). While this multiplier computes the (possibly denormal) product P , it is possible to perform the leading zero count on the subnormal input, the exponent sum update, and the computation of the shift distance. Then this information is

used to shift P properly (which may be back to normal or into the subnormal range) before rounding it. The details are left as an exercise for the reader; see [Mul+18] for a reference.

11.2.3 Faithful Floating-Point Multiplier

Using a faithful significand multiplier instead of the exact one entails an appreciable reduction in area and delay. It should be faithful to a $\text{ufix}(1, -w_F - 1)$ format (error on P strictly bounded by $2^{-w_F - 1}$):

- If $P \in [1, 2]$, then the rounding of P to an $\text{ufix}(1, -w_F)$ format adds an error bounded by $2^{-w_F - 1}$, and the sum of these two errors is bounded strictly by 2^{-w_F} : the resulting fraction is faithful.
- If $P \in [2, 4]$, the normalization divides by two the error of P but adds the error of dropping the bit of weight $2^{-w_F - 2}$: again the sum of these two errors and the rounding error is bounded strictly by 2^{-w_F} .

Therefore, the architecture of a faithful multiplier remains that in Fig. 11.7, with the P output becoming a $\text{ufix}(1, -w_F - 1)$ number.

We must also check that a faithful significand multiplier still always returns a result in $[1, 4]$; otherwise extra normalization hardware would be needed. Fortunately, the smallest possible product, 1, is exactly representable as a $\text{ufix}(1, -w_F - 1)$ number; therefore a faithful multiplier will return it. The largest exact product, as already seen, is $4 - 4 \cdot 2^{-w_F} + 2^{-2w_F}$. Adding to it the error bound $2^{-w_F - 1}$, we are still well below 4. This ensures that the product normalization implemented in Fig. 11.7 is still sufficient with a faithful multiplier.

11.2.4 Injection Rounding to Speed Up a Floating-Point Multiplier

Even with the baseline multiplier in Fig. 11.7, we have on the critical path two slow ($\log_2 w_F$ time) operations: the multiplier and the rounding adder. Can we fuse them into only one? The answer is yes, and the technique is known as *injection rounding*. The first idea is to inject a constant round bit of value $2^{-w_F - 1}$ in the partial product array of the multiplier, and we obtain $P' = P + 2^{-w_F - 1}$. This way, in the case when $P' \in [1, 2]$, just dropping the lower half of P' provides the result properly rounded to nearest with ties to away.

In the case when $P \in [2, 4)$, however, we should have added the rounding bit one position to the left. The same effect can be achieved by adding a second bit (correction bit) at the same position in this case.

To achieve this without adding the delay of a second addition, consider that P' is first obtained in carry save form (C, S) inside the multiplier (see Chap. 8), with a fast adder computing $C + S$ to complete the multiplication (only for the leading half of P' , since the lower bits will be discarded). The trick is again to use a compound adder to compute both $C + S$ and $C + S + 1$ (where the “+1” is the correction bit). Then a multiplexer controlled by the overflow bit of P' finally selects $C + S$ in the case $P' \in [1, 2)$ and $C + S + 1$ (shifted right one bit) in the case $P' \in [2, 4)$.

The interested reader will find in [Mul+18] a more detailed discussion of injection rounding that covers the other rounding modes (including round to nearest with ties to even) and subnormal support.

11.3 Floating-Point Division

Division and square root have one particularity: the division of normalized floating-point numbers is simpler than the division of fixed-point numbers, because it avoids the singularity of dividing by a number close to zero. Similarly, the square root of a normalized floating-point number avoids the singularity of \sqrt{x} near zero (where its derivative becomes infinite). Some fixed-point division and square root techniques first transform the inputs into some flavor of normalized floating point before actually computing the division.

Actual division algorithms are studied in Chap. 9. In this section, we focus on the management of the floating point in division. We also focus on hardware-oriented techniques. In software, the availability of a Fused Multiply-Add (FMA) allows for a wide range of efficient software-oriented techniques, well surveyed in two complementary textbooks [Mar00; CHT02].

11.3.1 Baseline Division Architecture

Special case management (summarized in Table 11.3) is comparable to that of multiplication.

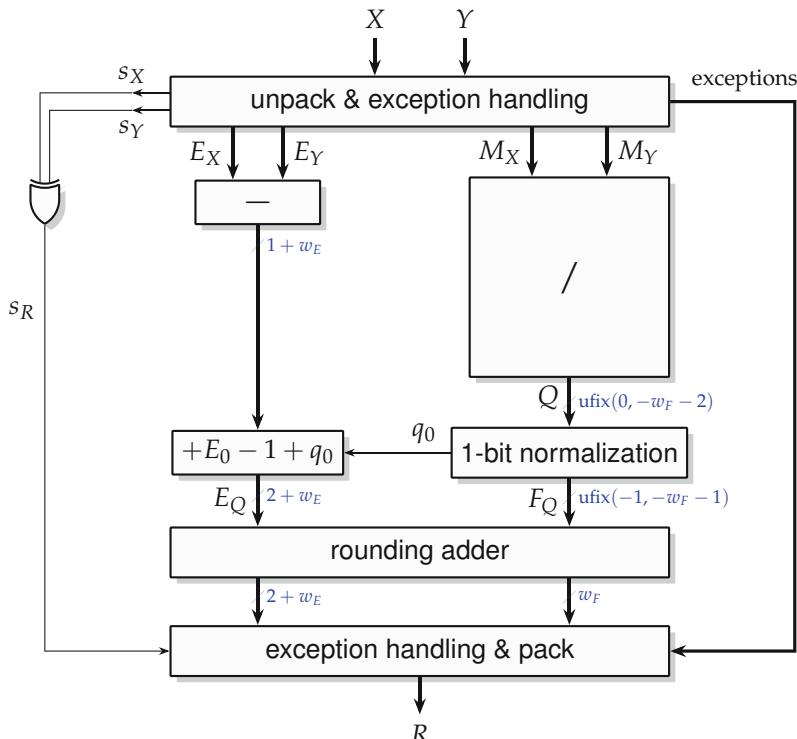
Table 11.3 Special cases of floating-point division for round to nearest.

/	NaN	-0	+0	$-\infty$	$+\infty$	$-\infty < X < 0$	$0 < X < +\infty$
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
-0	NaN	NaN	NaN	$-\infty$	$+\infty$	$+\infty$	$-\infty$
+0	NaN	NaN	NaN	$+\infty$	$-\infty$	$-\infty$	$+\infty$
$-\infty$	NaN	+0	-0	NaN	NaN	+0	-0
$+\infty$	NaN	-0	+0	NaN	NaN	-0	+0
$-\infty < Y < 0$	NaN	+0	-0	$+\infty$	$-\infty$	$\lfloor X/Y \rfloor$	$\lceil X/Y \rceil$
$0 < Y < +\infty$	NaN	-0	+0	$-\infty$	$+\infty$	$\lceil X/Y \rceil$	$\lfloor X/Y \rfloor$

Let us focus on the bottom-right corner of this table, first for normal numbers. The exact quotient of two normal numbers $X = (-1)^{s_X} \cdot 2^{E_X - E_0} \cdot M_X$ and $Y = (-1)^{s_Y} \cdot 2^{E_Y - E_0} \cdot M_Y$ is equal to

$$X/Y = (-1)^{s_X+s_Y} \cdot 2^{E_X - E_Y} \cdot M_X/M_Y. \quad (11.2)$$

Figure 11.8 illustrates the implementation of this operation. The exponent of the result is simply the difference of the input exponents, which is also equal to the difference of the biased exponent fields. However, to obtain a biased exponent field, the bias E_0 must be added back to the difference.

**Fig. 11.8** Baseline floating-point divider.

With $M_X \in [1, 2)$ and $M_Y \in [1, 2)$ (normalized inputs), we have $Q = M_X/M_Y \in (\frac{1}{2}, 2)$. In the case when $Q \in (\frac{1}{2}, 1)$ (i.e., the leading bit q_0 of Q is a 0), Q needs to be normalized:

- The mantissa Q is multiplied by 2 (1-bit left shift). This is what the **1-bit normalization** block does. This block also drops the leading 1 of the normalized quotient, keeping only its fraction F_Q .
- 1 must be subtracted to the exponent difference. This subtraction can be merged with the addition of E_0 : instead of adding E_0 , the architecture adds $E_0 - 1 + q_0$. Since $E_0 = 2^{w_E-1} - 1$, its LSB is a 1; hence the LSB of $E_0 - 1$ is a 0. The value $E_0 - 1 + q_0$ is therefore obtained by replacing the LSB of E_0 with q_0 .

The two adders on the exponent path on the left in Fig. 11.8 are best kept exact: the two MSB bits they add will be interpreted by the logic in the **exception handling and pack** block to decide underflow and overflow (in a way that depends on the floating-point format used).

Let us now address rounding. It is performed in two steps: firstly, inside the $\lceil \rceil$ block, the remainder is used to return in Q a truncation of the exact quotient to a $\text{ufix}(0, -w_F - 2)$ format. The extra LSB bit of weight $-w_F - 2$ is a guard bit that will become the round bit at position $-w_F - 1$ in the case $Q \in (\frac{1}{2}, 1)$.

The second step comes after normalization and is performed in the **rounding adder** block. This block concatenates F_Q and E_Q and then adds a rounding bit that is simply the bit of F_Q at position $-w_F - 1$. The rounding of the fraction may entail a carry-out that propagates to the exponent field. This implements round to nearest with ties to away, but ties are impossible in binary floating-point division with identical input and output sizes [Mul+18, lemma 4.13 p.130].

Note that $Q \in (\frac{1}{2}, 1)$ if and only if $M_X < M_Y$. The normalization of Q can therefore be predicted early by comparing M_X and M_Y , and the **1-bit normalization** block can be avoided altogether by, e.g., shifting M_X one bit to the left in this case. This alternative seems more expensive, but it can reduce latency if the $\lceil \rceil$ component involves some preprocessing such as prescaling [Bru18].

Hands on: Baseline floating-point division

The following command implements a floating-point divider for the Nfloat(8,23) format.

```
flopoco FPDiv we=8 wf=23
```

11.3.2 From Baseline Divider to IEEE 754 Compliance

11.3.2.1 Other Rounding Modes

The baseline architecture already performs round to nearest with ties to whatever you want, since ties cannot happen.

The support of directed rounding modes along with the round to nearest ones requires only small changes to the architecture. Ties for directed rounding may happen; they are exactly representable quotients (for instance, $1.0/1.0$). Whatever the rounding mode, if the exact result is already a floating-point number, this number must be returned. This was automatic in round to nearest but must be managed explicitly in directed rounding modes. An exact quotient corresponds to a null remainder, which can be computed (as the logical OR of all the remainder bits) inside the \square block, and output as a single bit, traditionally called “sticky bit” as in the adder and multiplier.

With this extra bit, directed rounding modes are straightforward to implement.

11.3.2.2 Subnormal Handling

Subnormal handling has a price (both in area and latency) comparable to what it has in multipliers. Here also, both input and output subnormals must be managed. Contrary to multiplication, the case when both inputs are subnormals must be managed. For standard formats, the quotient in this case will never be subnormal. Also note that most dividers studied in Chap. 9 require M_Y to be a normal number—again contrary to a mantissa multiplier, which works just as well if the leading bit of any of its input is not a 1.

The simplest option is to normalize inputs to an internal format with a 1-bit larger exponent, before using the baseline architecture. This normalization of subnormal inputs involves a leading zero counter and one shifter.

Subnormal outputs can be predicted early out of the exponent difference, with an uncertainty of 1 bit due to the possible normalization back to the normal range. Such possibly subnormal outputs are called tiny results.

Predicting tiny results may help reduce the latency of the significand quotient computation when the subnormal result requires less precision. Besides this, the management of subnormal outputs requires a large shift and specific rounding logic. In [Bru18], subnormal inputs may add two cycles before the main computation, and tiny results may add one cycle at the end.

11.3.3 Faithful Floating-Point Division

As for the multiplier, the architecture in Fig. 11.8 returns a faithful result if the $\boxed{\square}$ block is itself faithful to its output format (absolute error strictly bounded by 2^{-w_F-2}). The normalization may scale this error bound to 2^{-w_F-1} , and the rounding adder adds another error bounded by 2^{w_F-1} ; therefore the absolute error on the final mantissa is strictly bounded by 2^{-w_F} : the floating-point divider is faithful. In the case when the rounding adder updates the exponent, the significand is divided by 2, and so is the attached error; therefore the bound remains valid.

What is important here is the error bound $|\delta_Q| = |Q - M_X/M_Y| < 2^{-w_F-2}$. Instead of a faithful divider that outputs its result Q on the minimal format $\text{ufix}(0, -w_F - 2)$, it may be more efficient in this case to have for Q a slightly wider format $\text{ufix}(0, l_Q)$, as long as the error bound holds. Section 9.3.1, p. 289 gives an example in the case when the quotient is computed out of a faithful reciprocal approximation and a faithful multiplier. In this case, the output format is a $\text{ufix}(0, -w_F - 3)$ format. The multiplier itself is faithful to this format, but due to the fact that one of its inputs (the reciprocal approximation) was not exact, the overall error bound is $|\delta_Q| = |Q - M_X/M_Y| < 2^{-w_F-2}$.

We must also check that a faithful significand divider does not risk to return a result outside of $[1/2, 2]$, which would require extra normalization hardware. Here the argument is to consider the exact values of the two extremal quotients, which are $1/2$ and $2 - 2^{-w_F}$. Both are exactly representable as $\text{ufix}(0, -w_F - 2)$ numbers, and therefore a faithful divider will return them.

If the significand division is based on multiplication by an approximation to the inverse, the $\boxed{\square}$ block ends with a multiplier. In this case, injection rounding (studied in Sect. 11.2.4, p. 346) can be used to fuse the rounding adder inside the compression tree of this multiplier.

11.3.4 Correct Rounding Out of a Faithfully Rounded Quotient

This technique, illustrated by Fig. 11.9, starts with a quotient approximation that is faithful not to the target format but to one bit more: we start with a normalized significand M_Q faithful to $\text{ufix}(0, -w_F - 1)$. Noting $M_{X/Y}$ the normalized significand of the exact quotient X/Y (i.e., $M_{X/Y} = \sigma M_X/M_Y$ with $\sigma = 1$ if $M_X/M_Y \geq 1$ and $\sigma = 2$ otherwise), this means

$$|M_Q - M_{X/Y}| < 2^{-w_F-1} . \quad (11.3)$$

Then there are two possible cases:

- If M_Q is already a ufix($0, -w_F$) number (if its LSB is a 0), then (11.3) guarantees that M_Q is $M_{X/Y}$ correctly rounded to the nearest (Fig. 11.9a).
- Otherwise (Fig. 11.9b), M_Q is a mid-point between two consecutive floating-point numbers, and the error bound (11.3) ensures that M_Q is one of these two numbers (either $M_Q - 2^{-w_F-1}$ or $M_Q + 2^{-w_F-1}$). The decision can be taken by computing the remainder $\sigma M_X - M_Q M_Y$ and considering its sign:
 - If $\sigma M_X - M_Q M_Y > 0$, then $\sigma M_X / M_Y > M_Q$, and therefore M_Q should be rounded up.
 - If $\sigma M_X - M_Q M_Y < 0$, then $\sigma M_X / M_Y < M_Q$, and therefore M_Q should be rounded down.

The case $\sigma M_X - M_Q M_Y = 0$ is in general handled by the tie rule, but it cannot happen in a floating-point divider when the input and output formats are identical.

This solution only requires one final multiplication $M_Q M_Y$, which may be much cheaper than directly attempting to compute a correctly rounded quotient. Besides, a full multiplication is not needed. Since $|M_Q - \sigma M_X / M_Y| < 2^{-w_F-1}$ and $M_Y < 2$, we have $|M_Q M_Y - \sigma M_X| < 2^{-w_F}$. Now consider that 2^{-w_F} is the ulp of σM_X . Therefore, in $M_Q M_Y - \sigma M_X$, the only bits of weight lower than 2^{-w_F} come from $M_Q M_Y$. Therefore, if $M_Q M_Y - \sigma M_X > 0$, the upper bits of $M_Q Y$ must be equal to σM_X . In short, all we need is a left-truncated multiplier that computes the product $M_Q M_Y$ up to the LSB of σM_X . If this bit is equal to that of σM_X , it means that $M_Q M_Y - \sigma M_X \geq 0$. If they are different, then $M_Q Y - X < 0$.

Still, we need this multiplication and addition (plus another rounding addition and some logic) to compute only one extra bit of accuracy (actually to transfer this accuracy to the final format). This is a practical illustration of a point made in Chap. 3: for the same accuracy, it may be cheaper to use a faithful result to a 1-bit wider format. If a division is hidden in a wider circuit computing just right, it may well be cheaper to directly use the faithful quotient M_Q , despite it being 1-bit wider.

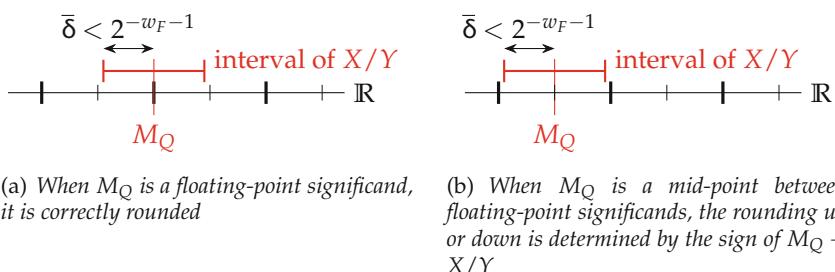


Fig. 11.9 The faithful rounding of the significand of X/Y to a ufix($0, -w_F - 1$) number M_Q enables its correct rounding to the nearest ufix($0, -w_F$) number.

A good overview of the state of the art in floating-point dividers for processors can be found in [Bru22]. It also covers square root, the subject of the next section.

11.4 Floating-Point Square Root

The square root function is one of the simplest operators to implement in floating point, firstly because it is a unary function and secondly because of the properties of the square root function itself.

11.4.1 Baseline Square Root Architecture

Special case management as per the IEEE 754 standard is summarized in Table 11.4. The IEEE 754 standard committee had one choice to make: should the square root of -0 be $+0$, -0 , or NaN? The choice made reflects that signed zeroes are still zeroes; they do not represent very small values (otherwise $\sqrt{-0}$ would be a NaN). The consequence is that an architecture can't just return a NaN as soon as the sign bit is set; it must also check for zero (Table 11.4).

Table 11.4 Special cases of floating-point square root for round to nearest.

X	NaN	-0	$+0$	$-\infty$	$+\infty$	$-\infty < X < 0$	$0 < X < +\infty$
\sqrt{X}	NaN	-0	$+0$	NaN	$+\infty$	NaN	$\lfloor \sqrt{X} \rfloor$

Let us now focus on the square root of a positive normal number $X = 2^{E_X - E_0} \cdot M_X$.

If $E_X - E_0$ is even, the square root is simply

$$\sqrt{X} = 2^{(E_X - E_0)/2} \times \sqrt{1.F_X} \quad (11.4)$$

If $E_X - E_0$ is odd, the previous formula would not give an integer exponent; therefore the following one is preferred:

$$\sqrt{X} = 2^{(E_X - E_0 - 1)/2} \times \sqrt{2 \times 1.F_X} \quad (11.5)$$

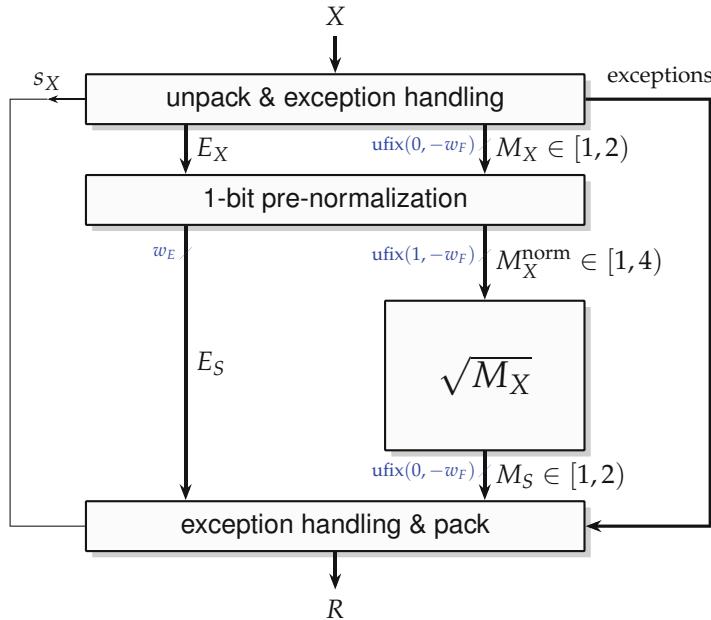


Fig. 11.10 Baseline floating-point square root architecture.

Figure 11.10 therefore begins with a pre-normalization that ensures that the exponent computed on the left is an integer. It is controlled by the lower bit e (for even) of the exponent field:

- Since E_0 is odd, $e = 1$ (odd exponent field E_X) means an even exponent $E_X - E_0$.
- On the significand side, the possible multiplication by 2 is a 1-bit shift, implemented as a multiplexer controlled by e .
- On the exponent side, we also need to add back E_0 to obtain the biased exponent field of the result E_S . Therefore we need to compute $E_0 + (E_X - E_0)/2$ if $e = 1$ and $E_0 + (E_X - E_0 - 1)/2$ if $e = 0$. This computation can be merged in a single addition:

$$E_S = \lfloor E_X/2 \rfloor + \lfloor E_0/2 \rfloor + e \quad (11.6)$$

where $\lfloor E_X/2 \rfloor$ consists of the $w_E - 1$ upper bits of the exponent, $\lfloor E_0/2 \rfloor$ is equal to $2^{w_E-2} - 1$ (a string of $w_E - 2$ ones), and e is a carry-in.

Then the $\sqrt{M_X}$ block computes the square root M_S of a number in the interval $[1, 4]$. Here M_S is directly in the expected format: there is no need for rounding logic (besides the rounding hidden in the $\sqrt{M_X}$ block, of course). There is also no need for any normalization logic, since a correctly rounded M_S is necessarily already normalized.

Exception handling is also quite simple, as the square root never overflows nor underflows (for instance, E_S can always be computed exactly on w_E bits).

Hands on: Baseline floating-point square root

The following command implements a floating-point square root for the Nfloat(8,23) format.

```
flopoco FPSqrt we=8 wf=23
```

11.4.2 Faithful Floating-Point Square Root

If the $\lceil \sqrt{M_X} \rceil$ block is faithful, an issue is that its result may need to be normalized. More precisely, the issue occurs for the largest possible value of M_X^{norm} , which is $M_X^{\max} = 2 \times (2 - 2^{-w_F})$. The Taylor development of the square root indicates that $\sqrt{M_X^{\max}} \approx 2 - 2^{-w_F-1} - 2^{-2w_F-4}$, very close to a midpoint between two floating-point mantissa. A faithful approximation (with an error bound 2^{-w_F}) may therefore return $M_S = 2.0$, which is not representable in the ufix($0, -w_F$) format of M_S seen on Fig. 11.10.

M_X^{\max} is the only value of M_X^{norm} with this problem. The faithful implementation of the $\lceil \sqrt{M_X} \rceil$ block may well, by luck, return in this case $M_S = 2 - 2^{-w_F}$ (the correctly rounded value), in which case the architecture is correct. However, it is important to check for it, because this single value is easily overlooked by random testing, and the consequence may be catastrophic: typically, the architecture of Fig. 11.10 will ignore the bit of weight 2^1 and return 1.0 for $\sqrt{4 - 2^{-w_F+1}}$ (and similarly wrong values for all the $4^i M_X^{\max}$ for integer i).

Now that our reader is aware of this issue, it is easy to correct, either by an ad hoc fix that makes sure that the value returned for $\lceil \sqrt{M_X^{\max}} \rceil$ is $2 - 2^{-w_F}$, or by saturation inside the $\lceil \sqrt{M_X} \rceil$ block, or by an explicit normalization step. In this case, Fig. 11.10 should be modified as follows:

- The format of M_S becomes ufix($1, -w_F$).
- Its MSB (the extra bit of position 1) is input to a multiplexer which selects between E_S and $E_S + 1$. The value of $E_S + 1$ can be precomputed in parallel to the $\lceil \sqrt{M_X} \rceil$ block.

Since this hardware only manages one single case, it remains lightweight; in particular there is no need to have a multiplexer on the significand side. There is also no need to add an extra rounding step after this normalization as we had in the multiplier or divider.

11.4.3 From Baseline Square Root to IEEE 754 Compliance

11.4.3.1 Tie Rule

Ties for round to nearest cannot happen in the square root of normal numbers, and therefore the architecture in Fig. 11.10 rounds to nearest whatever the tie rule. Indeed, consider an exact midpoint $S = M + 2^{-w_F-1}$ with M in ufix($0, -w_F$). Its square is $S^2 = M^2 + 2^{-w_F}M + 2^{-2w_F-2}$. There, M^2 is a ufix($1, -2w_F$) number (with a leading 1 in position 0 or 1), and $2^{-w_F}M$ is a ufix($-w_F, -2w_F$) number. We therefore have in S^2 a bit in position 0 or 1 and one bit in position $-2w_F - 2$. Therefore S^2 cannot be a floating-point number with a mantissa of $w_F + 1$ bits.

11.4.3.2 Other Rounding Modes

Rounding down and rounding to zero require changes only inside the $\boxed{\sqrt{M_X}}$ and $\boxed{\text{exception handling}}$ blocks. Conversely, the support of rounding up requires the small normalization unit described at the end of Sect. 11.4.2 to manage the single case $M_X^{\text{norm}} = M_X^{\max}$.

11.4.3.3 Subnormal Handling

The square root of a floating-point input is never in the subnormal range; therefore subnormal outputs cannot happen. A consequence is that ties for round to nearest never happen for subnormal inputs.

Conversely, subnormal inputs must be managed. The simplest solution is to convert subnormal inputs to normal numbers in an internal format with the same fraction size, but a 1-bit wider exponent. Since the exponent is going to be divided by two anyway, the baseline architecture can then be used without change. This normalization of subnormal inputs involves a leading zero counter and one shifter.

11.4.4 Correct Rounding Out of Faithful Rounding for Square Root

The technique of Sect. 11.3.4 also works for square root: just replace X/Y with \sqrt{X} in Fig. 11.9, and determine the sign of $S_0 - \sqrt{X}$ as that of $S_0^2 - X$. In this case, all we need to decide rounding is a left-truncated squarer [Din+10].

Actually, this technique works for any other algebraic functions. It will be exposed in full generality in Sect. 16.3.1.

However, the key message here remains that this last bit of accuracy may be quite expensive compared to faithful rounding.

11.5 Floating-Point Comparison

The comparison of normal floating-point numbers is greatly simplified by the biased encoding of the exponent. As already mentioned on p. 51, the order of positive floating-point numbers, from 0 to $+\infty$, is the order of their binary representation when interpreted as an integer. See also Table 2.1, p. 51. This even holds for subnormals: in a comparator, subnormal handling will be for free, and we will not mention this question further.

11.5.1 Specification of Floating-Point Comparison

It remains to decide what to do about special cases (signed zeroes, infinities, and NaNs). This question has been thought out in detail in the IEEE 754 standard [754-19]. The present section addresses the construction of a standard-compliant floating-point comparator, whose interface is given in Fig. 11.11. It outputs **four mutually exclusive comparison predicates** $X > Y$, $X = Y$, $X < Y$, and $X?Y$. The three first $X > Y$, $X = Y$, $X < Y$, are the standard mathematical comparison operators and are well defined on any finite input. With the additional specification that

- $+0$ and -0 compare equal,
- two infinities of the same sign compare equal,³

these three first predicates are well defined on all the floating-point numbers, except NaN (which are not numbers indeed).

The fourth predicate, $X?Y$, is defined in the IEEE 754 standard and reads *unordered*. It is true if and only if at least one of the inputs is a NaN.

³ Mathematically, the equality of two infinities could be endlessly debated. At least this choice is consistent with a comparison of the concatenation of fraction and exponent field (when using IEEE 754 encoding).

Table 11.6 Special cases for the floating-point comparison $X = Y$. The entry labeled $=$ represents the bitwise equality of the complete binary representations.

$X = Y$	$-\infty$	-0	$+0$	$ X > 0$	$+\infty$	NaN
$-\infty$	True	False	False	False	False	False
-0	False	True	True	False	False	False
$+0$	False	True	True	False	False	False
$ Y > 0$	False	False	False	=	False	False
$+\infty$	False	False	False	False	True	False
NaN	False	False	False	False	False	False

11.5.2 Implementation of a Floating-Point Comparator

The architecture of a comparator is straightforward (Fig. 11.12). It requires to compute “is Zero,” “is Infinity,” and “is NaN” signals for both inputs (this computation is the only difference between a comparator of IEEEfloat data and a comparator of Nfloat data). Then, it concatenates the exponent field E_X and the fraction field F_X to build an integer I_X and similarly for Y . An integer comparator computes three Boolean signals $I_X < I_Y$, $I_X > I_Y$, and $I_X = I_Y$. Finally, some logic must implement Tables 11.5 and 11.6 out of all the previous signals.

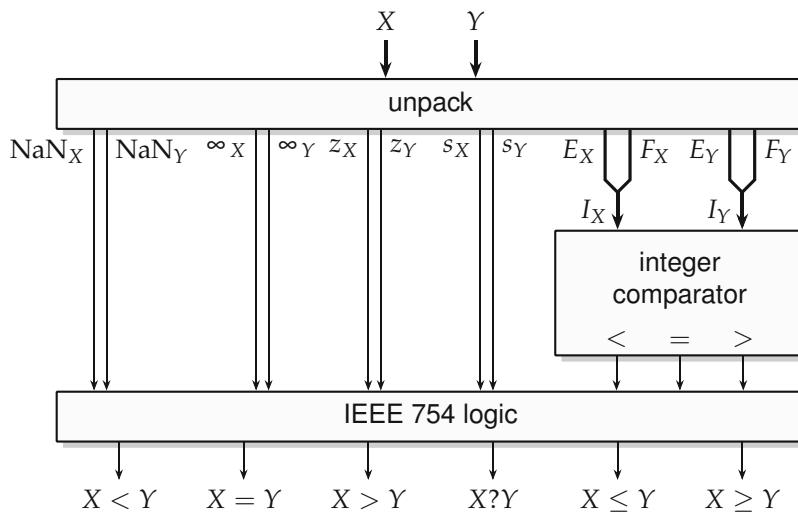


Fig. 11.12 Floating-point comparator architecture.

Hands on: Floating-point comparator

The following command implements a floating-point comparator for the Nfloat(8,23) format.

```
flopoco FPComparator we=8 wf=23
```

See the documentation of FPComparator for its other parameters.

11.5.3 Specializations of a Floating-Point Comparator

An application-specific comparator may require to implement one specific comparison, e.g., $X > Y$, in which case the hardware only used for the other comparisons can be trivially saved. However, this does not necessarily entail any saving in the integer comparator because of the necessity to compare signed numbers.

When the sign of both inputs is fixed, only one of the $I_X < I_Y$ and $I_X > I_Y$ comparator outputs is needed.

References

- [754-19] IEEE Standard for Floating-Point Arithmetic. also IEEE/ISO/IEC 60559-2020. 2019 (cit. on pp. [7](#), [330](#), [332](#), [357](#)).
- [Bru18] Javier D. Bruguera. “Radix-64 Floating-Point Divider”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2018, pp. 87–94 (cit. on pp. [4](#), [288](#), [349](#), [350](#)).
- [Bru22] Javier D. Bruguera. “Low-Latency and High-Bandwidth Pipelined Radix-64 Division and Square Root Unit”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2022 (cit. on pp. [4](#), [353](#)).
- [CHT02] Marius Cornea, John Harrison, and Ping Tak Peter Tang. *Scientific Computing on Itanium®-Based Systems*. Intel Press, 2002 (cit. on pp. [2](#), [294](#), [347](#)).
- [Din+10] Florent de Dinechin, Mioara Joldes, Bogdan Pasca, and Guillaume Revy. “Multiplicative square root algorithms for FPGAs”. In: *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2010, pp. 574–577 (cit. on pp. [13](#), [303](#), [307](#), [356](#)).
- [EL11] Pedro Echeverría and Marisa López-Vallejo. “Customizing floating-point units for FPGAs: Area-performance-standard

- trade-offs". In: *Microprocessors and Microsystems* 35.6 (2011), pp. 535–546 (cit. on p. [330](#)).
- [Lut17] David R. Lutz. "Optimized Leading Zero Anticipators for Faster Fused Multiply-Adds". In: *Asilomar Conference on Signals, Circuits and Systems*. IEEE, 2017, pp. 741–744 (cit. on p. [342](#)).
- [Lut19] David R. Lutz. "ARM Floating-Point 2019: Latency, Area, Power". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2019, pp. 69–76 (cit. on p. [342](#)).
- [Mar00] Peter Markstein. IA-64 and Elementary Functions: Speed and Precision. Hewlett-Packard Professional Books. Prentice Hall, 2000 (cit. on pp. [4](#), [25](#), [260](#), [288](#), [347](#)).
- [Mul+18] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. Handbook of Floating-Point Arithmetic. 2nd ed. Birkhäuser Boston, 2018 (cit. on pp. [11](#), [24](#), [311](#), [314](#), [324](#), [330](#), [332](#), [336](#), [346](#), [347](#), [349](#)).
- [SN01] Martin M. Schmookler and Kevin J. Nowka. "Leading Zero Anticipation and Detection - A comparison of methods". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2001, pp. 7–12 (cit. on pp. [322](#), [342](#)).
- [Soh+22] Jongwook Sohn, David K. Dean, Eric Quintana, and Wing Shek Wong. "Enhanced Floating-Point Adder with Full De-normal Support". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2022 (cit. on p. [342](#)).

Part II

Operator Specialization



12

CHAPTER 12

Multiplication by Constants

*Being unable to trust my reasoning, I learnt by heart
all the possible results of all the possible multiplications.*

Eugène Ionesco, *La leçon*

Multiplication by a constant is probably the most useful and best studied case of operator specialization, in particular for its importance in the construction of digital filters. There are two main families of constant multiplication techniques (shift-and-add and table-based), and there are several types of constants (integer/fixed-point, rational, or arbitrary real numbers). This chapter reviews the techniques that have been developed for each case. Some variants of the multiple constant multiplication problems are also addressed.

This chapter addresses the construction of an operator, depicted in Fig. 12.1, which multiplies an input X by a constant C hard-coded within the multiplier.

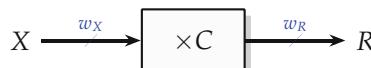


Fig. 12.1 Black-box interface to a constant multiplier.

A multiplier by a constant is potentially cheaper than a generic multiplier (with two inputs X and Y). For example, the multiplication by $C = 4$ of a binary integer is for free (it is a constant shift that can be realized purely by wiring). The multiplication by $C = 5$ of a binary integer X can be realized by one addition computing $X + 4X$, where $4X$ is again for free.

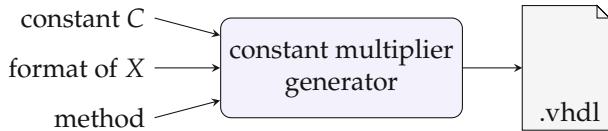


Fig. 12.2 Interface to a constant multiplier generator.

These are very simple examples, just to illustrate that the best architecture for multiplying by a given constant C strongly depends on the value of C . It also depends on the format of the input X . Actually, the best architecture should be computed out of C and the format of X , as illustrated by Fig. 12.2. This is the subject of this chapter.

There are two main techniques to realize constant multiplications:

- Shift-and-add techniques base the constant multiplication on a graph of additions, subtractions, and bit-shift operations. They are introduced in Sect. 12.1 in the simple case of integer C and X .
- The Ken Chapman's Multiplier (KCM) method tabulates subproducts in look-up tables (LUTs) and is thus more specific to FPGAs. It is introduced in Sect. 12.2, also for integer C and X .

Which method is the best strongly depends on the context and on the parameters of the problem. For instance, the complexity of the shift-and-add algorithms strongly depends on the constant itself, while the complexity of the KCM method mainly depends on the input word size.

In many applications, the relevant problem is not to multiply by an integer C , but by a real number C . For instance, in a fast Fourier transform (FFT), the constants are sines and cosines of $\frac{k\pi}{n}$ for k and n being integers. Section 12.3 adapts the two methods to this problem.

Some constants also have specific properties that enable specific algorithms. For instance, Sect. 12.4 studies the multiplication by rational constants.

Many applications require the multiplication by several constants, in which case the cost can be reduced by sharing resources between these constant multipliers. This is the subject of Sect. 12.5.

Finally, Sect. 12.6 describes further FPGA-specific techniques.

12.1 Shift-and-Add Integer Constant Multiplication

The multiplication of an unsigned binary integer X by an unsigned binary integer $C = (c_{w_C-1} \dots c_1 c_0)$, with $c_i \in \{0, 1\}$ can be written as

$$X \times C = X \cdot \left(\sum_{i=0}^{w_C-1} 2^i c_i \right) = \sum_{i=0}^{w_C-1} 2^i \underbrace{X \cdot c_i}_{\text{partial product}} . \quad (12.1)$$

Hence, the partial products $X \cdot c_i$ (obtained by a bitwise AND-operation) are bit-shifted and added to obtain the final product. Now, as C is a constant bit vector, all bits c_i which are zero lead to zero partial products, and the corresponding adders can be removed. Thus, the number of required adders for the constant multiplication using this representation is equal to the number of nonzero elements in the binary representation of C (also called its Hamming weight), minus one. To illustrate this, consider a constant multiplication by $C = 93$. Its binary representation $C = 1011101_2$ has five ones. The shift-and-add computation of $93X = (2^6 + 2^4 + 2^3 + 2^2 + 1)X$ requires four adders, as illustrated in Fig. 12.3a. There, bit shifts to the left are indicated by left arrows.

In the following, the data flow graph corresponding to this circuit, in which each node represents an adder or subtractor and the edge weights represent the shifts, is called an *adder graph*.

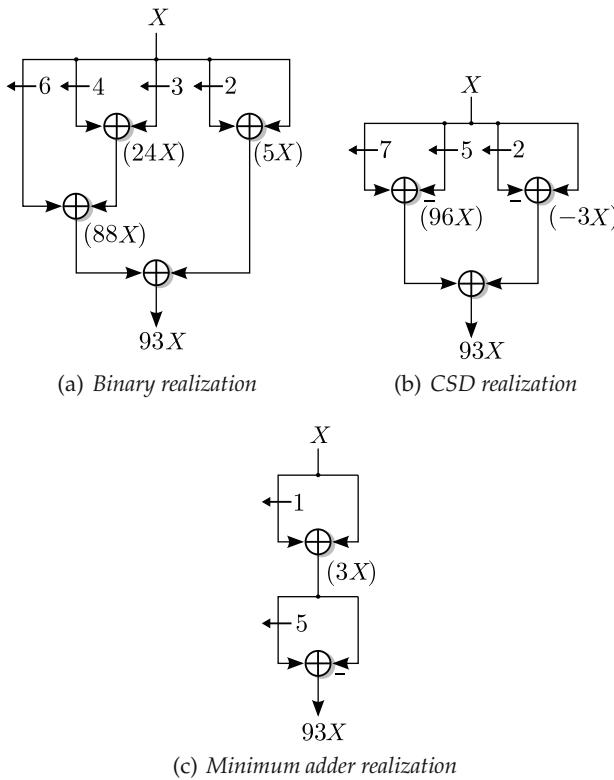


Fig. 12.3 Adder circuits to multiply by 93.

12.1.1 Signed Digit Representation

The binary representation is naturally limited to positive partial products. The total number of operations can be reduced in many cases by also allowing negative partial products, i.e., subtractions in the adder graph. For instance, $15X$ can be computed as $2^4 X - X$ instead of $2^3 X + 2^2 X + 2X + X$.

The subtract operation is nearly equivalent to the addition in hardware cost (in terms of area, speed, and energy). Therefore, in the following, both operations will be considered in adder graphs, and the term “adder” will cover both.

To exploit subtractions in adder graphs, the constant may be converted to the signed digit (SD) number system [Boo51; Toc58; Avi61] (sometimes also called “ternary balanced notation” [Knu97]). In this system, a constant C is still written $C = \sum_{i=0}^{w_C-1} 2^i c_i$, but each digit c_i can take the values $c_i \in \{-1, 0, 1\}$. For convenience, the value -1 is usually denoted as $\bar{1}$. In the example of Fig. 12.3, the coefficient can be represented by $93 = 10\bar{1}00\bar{1}01_{SD}$, i.e., $93 = 2^7 - 2^5 - 2^2 + 2^0$. Now, the corresponding circuit uses one adder less compared to the binary representation as illustrated in Fig. 12.3b.

Note that the signed digit number system is not unique as there may exist several alternative representations for the same number. For example, 93 could also be represented by $93 = 1\bar{1}01000\bar{1}1_{SD}$, which has the same number of nonzero digits as the binary representation. A unique representation with minimal number of nonzero digits is given by the canonical signed digit (CSD) representation. A binary number can be converted to CSD with the following simple algorithm [Mey14; Hwa79]:

Algorithm 12.1: CSD Conversion Algorithm

Starting with the least significant bit (LSB), search a bit string of the form ‘011…11’ with at least two consecutive ‘1’, and replace it with ‘10…01’ of the same length. This procedure is repeated until no further bit string can be found.

For example, the binary representation of $93 = 1011101_2$ is transformed to $93 = 1100\bar{1}01_{SD}$ in the first iteration by replacing ‘0111’ by $100\bar{1}$. Then, it is transformed to $93 = 10\bar{1}00\bar{1}01_{CSD}$ in the second and final iteration. The CSD representation is unique and guarantees a minimal number of nonzeros. Note that replacing strings of length two, i.e., ‘011’ by ‘10 $\bar{1}$ ’ as done in the last iteration does not change the adder count but replaces an adder by a subtractor (which may be unwanted). There may be other SD representations with a minimal number of nonzeros. They are called minimum signed digit (MSD) representations.

Starting from the CSD representation, valid MSD representations can be constructed by replacing the bit patterns ‘ $1\bar{0}\bar{1}$ ’ with ‘011’ or ‘ $\bar{1}01$ ’ with ‘ $\bar{0}\bar{1}\bar{1}$ ’. Doing this for all combinations results in a set of MSD numbers. For the constant 93, four different MSD representations can be constructed:

$$\begin{aligned} 93 &= 10\bar{1}00\bar{1}01_{\text{CSD}} \\ &= 01100\bar{1}01_{\text{MSD}} \\ &= 10\bar{1}000\bar{1}\bar{1}_{\text{MSD}} \\ &= 011000\bar{1}\bar{1}_{\text{MSD}} \end{aligned} \tag{12.2}$$

12.1.2 Formalization of the Shift-and-Add SCM Problem

Although the arithmetic complexity of the constant multiplier can be reduced by using an MSD representation of the constant, it is not guaranteed to be minimal. Consider again the example number 93: it can be factored into $93 = 3 \cdot 31$. With $3 = 11_{\text{MSD}}$ and $31 = 10000\bar{1}_{\text{MSD}}$, the cascade of these two constant multipliers reduces the required adders to only two as shown in Fig. 12.3c.

This solution can also be obtained by recognizing that two patterns in the CSD representation of 93 are related to each other:

$$10\bar{1}_{\text{SD}} = -\bar{1}01_{\text{SD}}. \tag{12.3}$$

With that, 93 can be represented as $93 = 10\bar{1}00\bar{1}01_{\text{CSD}} = 3 \cdot 2^5 - 3 = 3 \cdot (2^5 - 1) = 3 \cdot 31$. This concept is known as sub-expression sharing and the corresponding optimization method is called common subexpression elimination (CSE) [AJU76; PSC96; Har96]. However, there is no guarantee to find a solution with the minimal number of adders as some common patterns may be hidden by other patterns due to overlaps [FC10]. Take, for example, $25 = 10\bar{1}001_{\text{CSD}}$ which can be factored into $25 = 5 \cdot 5 = 101_2 \cdot 101_2$. Due to the fact that two ones in the pattern 101_2 overlap when computing $25 = 101_2 + 2^2 \cdot 101_2 = 11001_2$, the corresponding 101_2 pattern is not visible in the CSD representation.

Finding an adder circuit that multiplies with a given constant using a minimum number of adders is an optimization problem which is known as the single constant multiplication (SCM) problem. The SCM problem belongs to the class of NP-complete optimization problems [CS84]. However, for realistic coefficient word sizes (up to 32 bit), an optimal solution can often be found. In the following, we introduce two ways to find such optimal shift-and-add graphs: (1) by graph enumeration and (2) by using integer linear programming (ILP).

Before this is an important remark. There may be several minimal-adder graphs for a given constant, all with the same optimal number of adders. However, all these graphs may not lead to circuits with the same number of gates, or full adder (FA) cells. Section 12.1.6 will indeed show that the adders in a shift-and-add graph are of different sizes and will address the problem of obtaining a circuit minimizing the number of gates.

12.1.3 Minimal-Adder Constant Multiplication by Graph Enumeration

The basic idea in the graph enumeration approach is to somewhat reverse the problem by asking for the question “What coefficients can be computed using at most N_A adders?”. To answer this, all possible adder graph topologies using N_A adders are enumerated, with the corresponding constants computed and stored in a table. This table can then be searched to construct the optimal solution.

This graph enumeration approach was exploited by the minimized adder graph (MAG) algorithm of Dempster and Macleod for up to four adders. It was found to cover all 12 bit coefficients [DM94]. Later, Gustafsson et al. introduced simplifications that enhanced these results to five adders covering all coefficients up to 19 bit [GDW02; Gus+06]. Further extensions for up to six adders resulted in optimal SCM circuits for coefficients up to 32 bit [TN10; TN11]. In the latter work, not all graphs are generated exhaustively but the minimal representation of a given constant is obtained by the graph topology tests introduced by Voronenko and Püschel [VP07].

For illustration, Table 12.1 describes adder graphs for the first odd coefficients up to 8 bit, using the method proposed in [GDW02]. This table gives, for each coefficient, the number of adders (#+), the coefficients C_A and C_B , and shifts s_A and s_B to compute the coefficient using

$$C = C_A 2^{s_A} + C_B 2^{s_B}. \quad (12.4)$$

As long as C_A and/or C_B are not 1, the table has to be applied recursively on the nonzero C_A and/or C_B again.

Take, for instance, coefficient 173. The table tells us that it requires three additions and can be computed by $173 = 45 + 1 \cdot 2^7$. Next, coefficient 45 is looked up: it can be computed by $45 = 15 + 15 \cdot 2^1$. Finally, coefficient 15 can be computed from $C_A = C_B = 1$ using one addition, i. e., $15 = 1 \cdot 2^4 - 1$. Multiplying both sides by input value X leads to the following solution:

$$15X = X \cdot 2^4 - X \quad (12.5)$$

$$45X = 15X + 15X \cdot 2^1 \quad (12.6)$$

$$173X = 45X + X \cdot 2^7 \quad (12.7)$$

which is shown in Fig. 12.4.

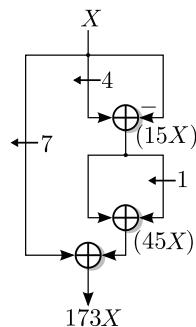


Fig. 12.4 Minimal-adder circuit to multiply with 173.

Note that an even 8-bit coefficient can be computed from an odd coefficient by applying one or more left shifts at the output. For that, the even coefficient is first divided by two (l times) until it is odd. The adder graph for the resulting odd coefficient is constructed from the table, and the result is left shifted by l bit. For example, coefficient 120 has to be divided three times by two to obtain the odd coefficient 15; hence, 120 can be computed by $120 = 2^3 \cdot 15$ using one addition (to compute $15 = 2^4 - 1$).

Hands on: Minimal-adder constant multipliers

The FloPoCo core generator defaults to least-adder shift and add for coefficients up to 19 bits. The following FloPoCo call generates VHDL code for the optimal (least adder) constant multiplier for the example constant 173 of Fig. 12.4 and an input word size of 10 bit:

```
flopoco IntConstMult method=minAdd wIn=10 constant=173
```

Table 12.1 Minimal-adder SCM solutions for coefficients up to 8 bit. To obtain a graph in #+ adders, recursively apply $C = C_A 2^{s_A} + C_B 2^{s_B}$ until both C_A and C_B are 1.

C	#+	C_A	s_A	C_B	s_B	C	#+	C_A	s_A	C_B	s_B	C	#+	C_A	s_A	C_B	s_B
3	1	1	0	1	1	89	3	1	0	11	3	175	3	11	4	-1	0
5	1	1	0	1	2	91	3	23	2	-1	0	177	3	1	0	11	4
7	1	1	3	-1	0	93	2	31	0	31	1	179	3	51	0	1	7
9	1	1	0	1	3	95	2	3	5	-1	0	181	3	1	0	45	2
11	2	1	3	3	0	97	2	1	0	3	5	183	3	23	3	-1	0
13	2	1	0	3	2	99	2	33	0	33	1	185	3	1	0	23	3
15	1	1	4	-1	0	101	3	1	0	25	2	187	3	47	2	-1	0
17	1	1	0	1	4	103	3	13	3	-1	0	189	2	63	0	63	1
19	2	3	0	1	4	105	2	15	3	-15	0	191	2	3	6	-1	0
21	2	1	0	5	2	107	3	1	7	-21	0	193	2	1	0	3	6
23	2	3	3	-1	0	109	3	1	7	-19	0	195	2	65	0	65	1
25	2	1	0	3	3	111	2	7	4	-1	0	197	3	1	0	49	2
27	2	1	5	-5	0	113	2	1	0	7	4	199	3	25	3	-1	0
29	2	1	5	-3	0	115	3	1	7	-13	0	201	3	1	0	25	3
31	1	1	5	-1	0	117	3	1	7	-11	0	203	3	51	2	-1	0
33	1	1	0	1	5	119	2	1	7	-9	0	205	3	1	0	51	2
35	2	3	0	1	5	121	2	1	7	-7	0	207	3	13	4	-1	0
37	2	5	0	1	5	123	2	1	7	-5	0	209	3	1	0	13	4
39	2	5	3	-1	0	125	2	1	7	-3	0	211	3	1	8	-45	0
41	2	1	0	5	3	127	1	1	7	-1	0	213	3	85	0	1	7
43	3	1	5	11	0	129	1	1	0	1	7	215	3	1	8	-41	0
45	2	15	0	15	1	131	2	3	0	1	7	217	2	31	3	-31	0
47	2	3	4	-1	0	133	2	5	0	1	7	219	3	1	8	-37	0
49	2	1	0	3	4	135	2	7	0	1	7	221	3	1	8	-35	0
51	2	17	0	17	1	137	2	9	0	1	7	223	2	7	5	-1	0
53	3	1	0	13	2	139	3	11	0	1	7	225	2	1	0	7	5
55	2	1	6	-9	0	141	3	13	0	1	7	227	3	1	8	-29	0
57	2	1	0	7	3	143	2	9	4	-1	0	229	3	1	8	-27	0
59	2	1	6	-5	0	145	2	1	0	9	4	231	2	33	3	-33	0
61	2	1	6	-3	0	147	3	19	0	1	7	233	3	1	8	-23	0
63	1	1	6	-1	0	149	3	21	0	1	7	235	3	1	8	-21	0
65	1	1	0	1	6	151	3	23	0	1	7	237	3	1	8	-19	0
67	2	3	0	1	6	153	2	17	0	17	3	239	2	1	8	-17	0
69	2	5	0	1	6	155	2	31	0	31	2	241	2	1	0	15	4
71	2	9	3	-1	0	157	3	29	0	1	7	243	3	1	8	-13	0
73	2	1	0	9	3	159	2	5	5	-1	0	245	3	1	8	-11	0
75	2	15	0	15	2	161	2	1	0	5	5	247	2	1	8	-9	0
77	3	13	0	1	6	163	3	35	0	1	7	249	2	1	8	-7	0
79	2	5	4	-1	0	165	2	33	0	33	2	251	2	1	8	-5	0
81	2	1	0	5	4	167	3	21	3	-1	0	253	2	1	8	-3	0
83	3	19	0	1	6	169	3	1	0	21	3	255	1	1	8	-1	0
85	2	17	0	17	2	171	3	1	0	85	1						
87	3	11	3	-1	0	173	3	45	0	1	7						

12.1.4 Minimal-Adder Constant Multiplication by Using ILP

This section introduces an integer linear programming (ILP) formulation of the shift-and-add constant multiplication problem. For more detail about using ILP for optimization, we refer the reader to Appendix B. An ILP-based method may be slower than highly specialized branch-and-bound algorithms [AGF08; Aks09; AGF10], but it is easy to re-implement, and it provides a framework for various optimization objectives or additional constraints. Examples for different objectives are minimizing power consumption [DDK00; DDK02; Joh08; FC10; Aks+11; YY11], the combination with LUT-based multipliers [Kum+13a] or additional constraints on fan-out [Gus08], pipelining [Kum+12], ternary adders [Kum+13b; Kum+16], or re-configuration [Möl+17; Möl17]. Another advantage is that any progress in the performance of ILP solvers translates into the performance of solving the constant multiplication problem. Finally, it can also be easily extended to other problems such as multiple constant multiplication (MCM) [Kum18] (this will be the subject of Sect. 12.5) or digital filters [SY11] (see Chap. 23).

The objective of the SCM problem is to minimize the number of adders N_A . However, it is much easier to write an ILP formulation assuming that the number of adders N_A is known. Therefore, to find the actual minimal number of adders, the algorithm will start the search by attempting to solve the ILP for a known lower bound $N_{A,\text{lb}}$ of N_A and, if infeasible, increase N_A by one, until a feasible solution is found. As a consequence, we only have to find a feasible solution that fulfills the constraints. There is no objective set for each run of the ILP. This leaves the possibility of using the objective function for a secondary objective, for instance, power consumption [Kum18].

There are well-known lower bounds for the adder count available [Gus07b; TMJ14; JBD17]. When multiplying with a single constant (which we will extend later in Sect. 12.5 to multiple constants), the lower bound can be computed by the following idea: A single adder can compute at most two nonzero digits in the MSD representation of the constant c . Adding another adder can at most add two additional nonzero digits, etc. Hence, knowing the number of nonzeros of an MSD representation of c (like the CSD format obtained by Algorithm 12.1), denoted by $\text{nz}(c)$, we can compute the lower bound by

$$N_{A,\text{lb},\text{SCM}} = \lceil \log_2(\text{nz}(c)) \rceil. \quad (12.8)$$

We first start with a nonlinear mathematical model of the SCM problem to highlight the main ideas and the definition of the main variables. These nonlinear constraints are next linearized.

12.1.4.1 A First Nonlinear Model

Each adder node of an adder graph is labeled with an index $a = 1 \dots N_A$ in the following. Besides, we define an input node $a = 0$ and its corresponding constant is constrained to be

$$\mathcal{C}1 : \quad c_0 = 1 .$$

The constant of an adder node is computed from two other nodes by

$$c_a = (-1)^{\phi_{a,L}} 2^{s_{a,L}} c_{a,L} + (-1)^{\phi_{a,R}} 2^{s_{a,R}} c_{a,R}, \forall a = 1 \dots N_A \quad (12.9)$$

where $\phi_{a,L}$ (resp. $\phi_{a,R}$), $s_{a,L}$ (resp. $s_{a,R}$), and $c_{a,L}$ (resp. $c_{a,R}$) correspond to the sign, the bit shift, and the constant computed by the left (resp. right) input node of the adder.

Now, the SCM problem consists in finding a feasible solution of source nodes, shifts, and signs in (12.9) for a sequence of adder nodes, such that one coefficient c_a is identical to the target constant C . Again, the real objective is to minimize N_A , but in the ILP formulation N_A is a constant that determines the number of equations represented by (12.9).

As (12.9) is nonlinear, it cannot be used in standard ILP solvers. For that, a linearized model is derived in the following. This model will use so-called *indicator constraints* which are themselves not linear but can be easily linearized (see Appendix B.3.3). Indicator constraints contain a binary decision variable which controls whether or not a specified linear constraint is active. They are directly supported by modern ILP solvers. Typically, resolution of problems expressed using indicator constraints is numerically more robust than with big-M constraints. However, it may be slower due to weaker relaxations.

12.1.4.2 A Model Using Indicator Constraints

First, the nonlinear relation of (12.9) is split into several additional variables

$$c_a = \underbrace{(-1)^{\phi_{a,L}} \underbrace{2^{s_{a,L}} c_{a,L}}_{=c_{a,L}^{\text{sh}}} + (-1)^{\phi_{a,R}} \underbrace{2^{s_{a,R}} c_{a,R}}_{=c_{a,R}^{\text{sh}}}}_{=c_{a,L}^{\text{sh,sg}}} . \quad (12.10)$$

The constant after the bit shift of the left (L) and right (R) input, is represented by $c_{a,L}^{\text{sh}}$ (resp. $c_{a,R}^{\text{sh}}$), respectively, while $c_{a,L}^{\text{sh,sg}}$ (resp. $c_{a,R}^{\text{sh,sg}}$) also includes the sign. Now, the adder result can be represented using the linear expression

Table 12.2 Notations used in the SCM and MCM ILP formulation.

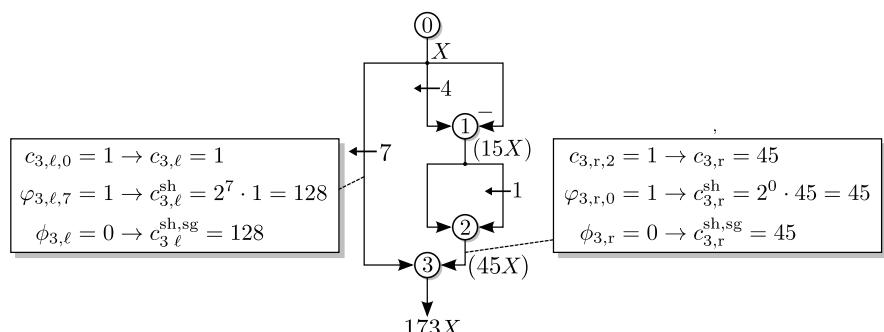
Constant	Meaning
$N_A \in \mathbb{N}$	Number of adders
C	Target constant coefficient
$S_{\min}, S_{\max} \in \mathbb{Z}$	Minimum and maximum shift
Variable	Meaning
$c_a \in \mathbb{N}$	Constant computed in adder a
$c_{a,i} \in \mathbb{N}$	Constant of input $i \in \{\text{L, R}\}$ of adder a
$c_{a,i}^{\text{sh}} \in \mathbb{N}$	Shifted constant of input $i \in \{\text{L, R}\}$ of adder a
$c_{a,i}^{\text{sh,sg}} \in \mathbb{N}$	Shifted, sign corrected constant of input $i \in \{\text{L, R}\}$ of adder a
$\phi_{a,i} \in \{0, 1\}$	Sign of input $i \in \{\text{L, R}\}$ of adder a ($0: '+'$, $1: '-'$)
$c_{a,i,k} \in \{0, 1\}$	True, if input i of adder a is connected to adder k
$\varphi_{a,i,s} \in \{0, 1\}$	True, if input i of adder a is shifted by s bits

$$\mathcal{C}2 : \quad c_a = c_{a,\text{L}}^{\text{sh,sg}} + c_{a,\text{R}}^{\text{sh,sg}} \quad \forall a = 1 \dots N_A .$$

Next, several binary decision variables are used to select the sources, shifts, and signs of each node. An overview of the constants and variables (including the ones explained next) is given in Table 12.2. All variables describing the configuration of the adder computing $173x$ in Fig. 12.4 are shown in Fig. 12.5.

12.1.4.3 Adder Source

The binary variable $c_{a,i,k}$ is used to select the source. It is defined to be true, when input $i \in \{\text{L, R}\}$ of adder a is connected to adder k . This can be represented by the following constraints:

**Fig. 12.5** Variables for the output adder computing $173X$ of the example in Fig. 12.4.

$$\mathcal{C}3a : \quad c_{a,i} = c_k \text{ if } c_{a,i,k} = 1 \quad \forall a = 1 \dots N_A, i \in \{\text{L, R}\}$$

$$k = 0 \dots a - 1$$

$$\mathcal{C}3b : \quad \sum_{k=1}^{a-1} c_{a,i,k} = 1 \quad \forall a = 1 \dots N_A, i \in \{\text{L, R}\}$$

$\mathcal{C}3a$ is an indicator constraint (there is an “if” in it). It assigns the value c_k in case $c_{a,i,k}$ is set. The $\mathcal{C}3b$ constraint makes sure that exactly one source is selected for each adder input.

Instead of allowing arbitrary connections from any adder to any other adder, the adders are ordered and only connections from previous adders with a lower id (i.e., $k < a$) are allowed. This has two benefits. First, loops are prevented, i.e., factors which are computed from succeeding adders forming a loop. Second, the search space is reduced, as many combinations having the same quality are excluded.

12.1.4.4 Shift of Input

The binary variable $\varphi_{a,i,s}$ is used to select the bit shift. It is defined to be true, when input $i \in \{\text{L, R}\}$ of adder a is shifted by s bits. With that, the shifted constant is obtained by

$$\mathcal{C}4a : \quad c_{a,i}^{\text{sh}} = 2^s c_{a,i} \text{ if } \varphi_{a,i,s} = 1 \quad \forall a = 1 \dots N_A, i \in \{\text{L, R}\},$$

$$s = S_{\min} \dots S_{\max}$$

$$\mathcal{C}4b : \quad \sum_{s=0}^{S_{\max}-1} \varphi_{a,i,s} = 1 \quad \forall a = 1 \dots N_A, i \in \{\text{L, R}\}$$

Again, indicator constraint $\mathcal{C}4a$ applies the shift while $\mathcal{C}4b$ ensures that exactly one shift is selected. However, not all possible shift combinations are necessary. In general, it is sufficient that one input has a positive (left) shift, while the other is zero, or both inputs have an identical negative (right) shift [Kum15]. As the inputs can be arbitrarily connected in $\mathcal{C}3$, we can further tighten the formulation by allowing only the left input shift to be nonzero positive and requiring both shifts to be identical for negative shifts

$$\mathcal{C}4c : \quad \varphi_{a,R,s} = 0 \quad \forall s > 0$$

$$\mathcal{C}4d : \quad \varphi_{a,L,s} = \varphi_{a,R,s} \quad \forall s < 0.$$

12.1.4.5 Sign of Input

Another binary variable $\phi_{a,i}$ is defined for each adder input that determines the sign. The shifted and sign-corrected input of adder node a is obtained

by

$$\mathcal{C}5a : \quad c_{a,i}^{\text{sh,sg}} = -c_{a,i}^{\text{sh}} \text{ if } \phi_{a,i} = 1 \quad \forall a = 1 \dots N_A, i \in \{\text{L, R}\}$$

$$\mathcal{C}5b : \quad c_{a,i}^{\text{sh,sg}} = c_{a,i}^{\text{sh}} \text{ if } \phi_{a,i} = 0 \quad \forall a = 1 \dots N_A, i \in \{\text{L, R}\}$$

$$\mathcal{C}5c : \quad \phi_{a,\text{L}} + \phi_{a,\text{R}} \leq 1 \quad \forall a = 1 \dots N_A.$$

While $\mathcal{C}5a$ and $\mathcal{C}5b$ are obvious, constraints $\mathcal{C}5c$ ensure that only one input of the adder is subtracted (as subtracting both inputs is typically more hardware demanding).

12.1.4.6 Output Constraint

For the SCM problem, the last adder has to be constrained to the target coefficient C

$$\mathcal{C}6 : \quad c_{N_A} = C.$$

This constraint can be simply extended for the MCM problem later in Sect. 12.5.1.1.

12.1.4.7 Fully Linear ILP Model

As discussed in Appendix B.3.3, an indicator constraint of the form

$$x = y \text{ if } z = 1 \tag{12.11}$$

can be rewritten in linear form by using a big-M constraint

$$y - M + Mz \leq x \leq y + M - Mz. \tag{12.12}$$

Using this relation, the indicator constraints $\mathcal{C}3a$, $\mathcal{C}4a$, $\mathcal{C}5a$, $\mathcal{C}5b$, and $\mathcal{C}6a$ can be linearized as follows:

$$\mathcal{C}3a' : \quad c_k - M + Mc_{a,i,k} \leq c_{a,i} \leq c_k + M - Mc_{a,i,k}$$

$$\mathcal{C}4a' : \quad 2^s c_{a,i} - M + M\varphi_{a,i,s} \leq c_{a,i}^{\text{sh}} \leq 2^s c_{a,i} + M - M\varphi_{a,i,s}$$

$$\mathcal{C}5a' : \quad c_{a,i}^{\text{sh}} - M\phi_{a,i} \leq c_{a,i}^{\text{sh,sg}} \leq c_{a,i}^{\text{sh}} + M\phi_{a,i}$$

$$\mathcal{C}5b' : \quad -c_{a,i}^{\text{sh}} - M + M\varphi_{a,i} \leq c_{a,i}^{\text{sh,sg}} \leq -c_{a,i}^{\text{sh}} + M - M\varphi_{a,i}$$

The most critical constraint above is $\mathcal{C}4a'$. Here, M must be larger than the maximum of $c_a 2^s$. The shift is bounded by S_{\max} while the coefficient c_a can unfortunately be larger than C itself. A hardware implementation would

suffer from large intermediate constants c_a as the word size of the adders increases. Thus, it a good idea to set M to something like $M = 2C2^{S_{\max}}$ and to bound the c_a values by $c_a < 2C$.

12.1.4.8 Practical Limitations of the Model

This ILP model is currently limited to rather small problem sizes due to two scaling issues in the ILP solvers. The first is the runtime of the solver: It depends on the problem size, more specifically on the required number of adders (which tends to increase with the coefficient size). Another limitation comes from numerical issues in the solver. Large coefficients require large values of M in big- M constraints, which cause numerical problems and dictate the use of (slower) indicator constraints. This is discussed in more detail in Sect. B.4 of the Appendix.

In practice, constants up to 12 bits can typically be solved in less than one second on a current laptop machine using the Gurobi ILP solver with indicator constraints [Gurobi]. The runtime for 16-bit constants varies between sub-second and several minutes. It can already reach one hour for 20-bit constants. This limits the practical use to constant sizes of about 20 bits. For larger constants, heuristics should be used: They are discussed in Sect. 12.5.1 below.

12.1.5 Minimal-Adder Constant Multiplication Using Ternary Adders

As introduced in Sect. 5.4.4, ternary adders (adders with three inputs) can be efficiently mapped to field programmable gate arrays (FPGAs). Figure 12.6 shows the example of multiplication with constant 683, using 2-input adders as well as using ternary adders.

12.1.5.1 Results from Graph Enumeration

Following the idea of Sect. 12.1.3, all possible graph topologies using ternary adders can be enumerated and evaluated. The resulting topologies were evaluated in [Kum+16]. The rules to compute all related coefficients are more complicated than those using 2-input adders. Hence, we limit the following discussion to the results of [Kum+16], where the topologies of up to three ternary adders were evaluated. It turns out that all constants up to 22 bits can be computed with three ternary adders or fewer. To be more specific, constant 7,154,955 is the first number that requires more than three ternary adders. Figure 12.7a shows the average and maximal adder cost (up-

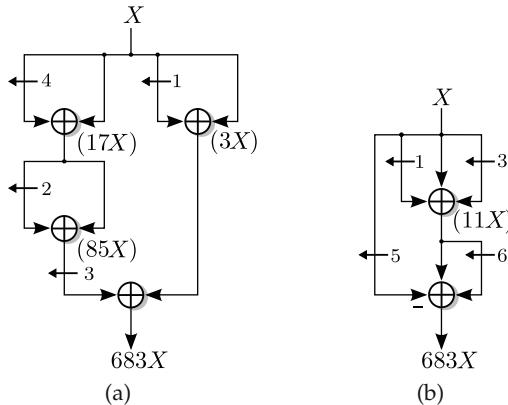


Fig. 12.6 Optimal SCM adder circuits which realize the multiplication with 683. (a) Using 2-input adders. (b) Using ternary adders.

per bound) using 2-input adders and ternary adders up to 19 bits (limit in [Gus+06]) and 22 bits, respectively. The adder cost improvement is illustrated in Fig. 12.7b. It can be observed that about 1/3 of the resources can be saved on average for coefficients with more than five bits by using ternary adders.

Hands on: Least-ternary-adder constant multipliers

The method was implemented for constant multiplier cores up to 22-bit coefficients within the FloPoCo core generator. The following FloPoCo call generates VHDL code of an optimal (least ternary adder) constant multiplier with constant 173 and an input word size of 10 bits:

```
flopoco IntConstMult method=minAddTernary wIn=10 \
    constant=173
```

12.1.5.2 Ternary Adder ILP Extension

Another way to optimize constant multipliers with ternary adders is the extension of the ILP formulation of Sect. 12.1.4. Constraints $\mathcal{C}2$ and $\mathcal{C}5c$ have to be replaced by

$$\begin{aligned} \mathcal{C}2^* : \quad c_a &= c_{a,L}^{\text{sh,sg}} + c_{a,M}^{\text{sh,sg}} + c_{a,R}^{\text{sh,sg}} \quad \forall a = 1 \dots N_A \\ \mathcal{C}5c^* : \quad \phi_{a,L} + \phi_{a,M} + \phi_{a,R} &\leq 2 \quad \forall a = 1 \dots N_A \end{aligned}$$

where the third input is represented as additional (middle) input $i = M$. While the extension in $\mathcal{C}2^*$ is obvious, $\mathcal{C}5c^*$ limits up to two of the three in-

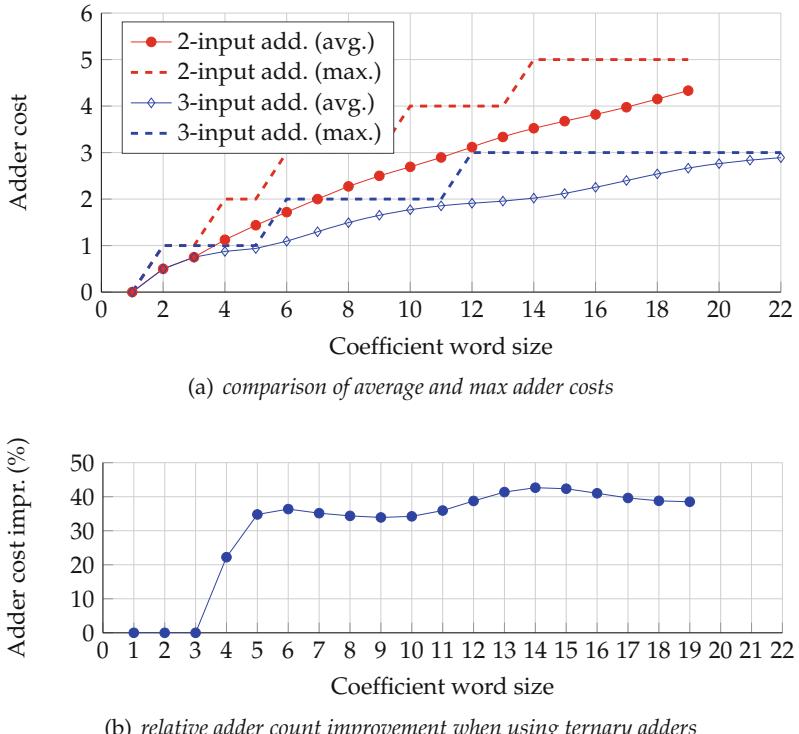


Fig. 12.7 Adder cost comparison between optimal SCM circuits using 2-input and ternary adders.

puts to being negative, which is realizable on modern FPGAs [Kum+13b]. Constraints $\mathcal{C}3a/b$, $\mathcal{C}4a/b$, and $\mathcal{C}5a/b$ have to be extended to include the case $i = M$, i. e., $i \in \{L, M, R\}$. Constraints $\mathcal{C}1$, $\mathcal{C}4c/d$, and $\mathcal{C}6$ remain identical.

12.1.6 Minimizing Logic Resources Instead of Number of Adders

The number of adders (incl. subtractors) is an important high-level metric, and it may be relevant for a software implementation. However, it does not reflect precisely the implementation cost. Figure 12.8 illustrates this when all the adders are realized as ripple-carry adders (RCAs).

Due to the shift, parts of the adders add zero bits. This is illustrated in Fig. 12.8b. On the LSB, the corresponding FA cells can be saved. On the MSB, they can be replaced with cheaper half adder (HA) cells.

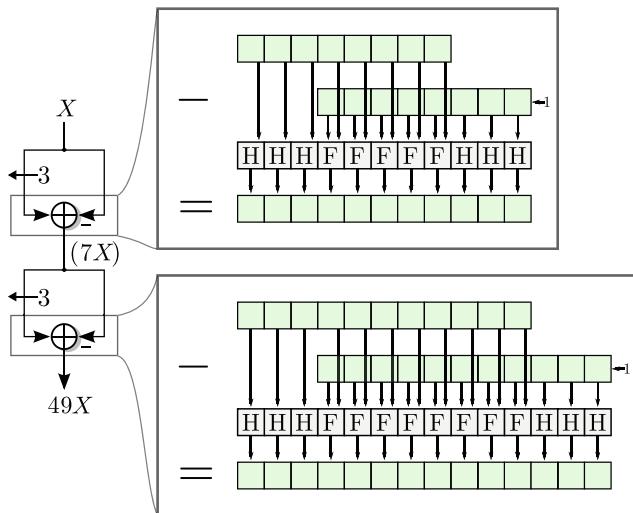
Here the number of FAs and HAs in each adder is not directly related to the word size of the constant. The type of operation (add/subtract) and the bit shifts of their arguments influence the FA cost. There are typically many possible ways to compute a constant multiplication using the minimum number of adders, and they may differ in bit-level complexity.

Figure 12.8 shows an example of two minimal-adder (two adders) SCM circuits computing $49X$. Their bit-level implementation is shown on the right for an 8-bit X . Input and output bits are represented by green boxes while the FAs and FAs are represented by “F” and “H,” respectively. In the following, we will count both as FA equivalents to simplify comparisons, as an FA can be used to compute an HA. In case of subtraction, negative inputs have to be two’s complemented, which requires an inversion (not shown in the figures) and an increment, hence the HAs at the LSB.

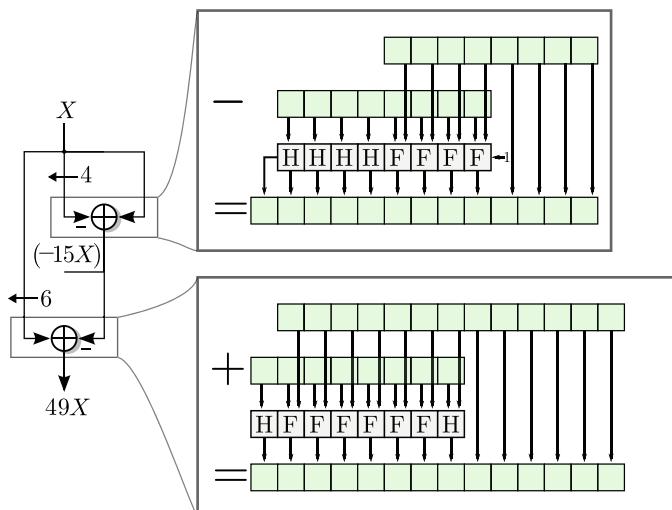
Figure 12.8a uses a rather small intermediate constant of $7X$ to compute $49X$. Due to the negation of the non-shifted input, full adder equivalents are required from LSB to most significant bit (MSB). It requires 11 full adder equivalents to compute $7X$ and another 14 full adder equivalents to compute $49X$, leading to 25 full adder equivalents. In contrast, Fig. 12.8b shows an SCM circuit designed for minimum number of full adder equivalents. It uses the negative intermediate constant $-15X$ to compute $49X$. Computing $-15X$ allows to copy four LSBs which requires only 8 full adder equivalents. This allows to use an addition in the following adder that also requires only 8 full adder equivalents, leading to 16 full adder equivalents in total (about 1/3 less).

Bit-level costs were first proposed by Johansson et al. [JGW05; JGW07] and later used by Aksoy et al. [Aks+07] and Brisebarre et al. [BDM08]. Their models consider the bit shifts of all the cases that can occur in an adder graph leading to an accurate number of full adders, half adders, and even inverters [Aks+07]. The beneficial use of negative constants was highlighted in [GJD09]. A bit-level model can also be used to reduce the critical path delay as demonstrated by Lou et al. [LYM14; LYM15].

An ILP extension to achieve the minimum full adders can be found in [GVK22; GV23]. It generalizes the ILP formulation presented in Sect. 12.1.4 by introducing constraints to model a detailed cost metric at the level of full adders.



(a) An arbitrary minimal-adder solution



(b) Minimum full adder solution

Fig. 12.8 Bit-level cost of multiplying by 49 using 2 adders.

12.2 Integer Constant Multiplication Using Tables

As most FPGAs base their logic fabric on a large number of small lookup tables (LUTs), this chapter surveys ways of exploiting such LUTs in constant multiplication algorithms. This section is therefore quite specific to FPGAs.

In all this section, the parameter α denotes the number of inputs of the architectural LUT of the target FPGA. For instance, on mainstream FPGAs at the time of writing, α ranges from 4 to 6.

12.2.1 Tabulated Constant Multipliers

The input X can only take a finite number (exactly 2^{w_X}) of possible values. It is therefore possible in theory to build a multiplier by simply tabulating all the possible products, as illustrated by Fig. 12.9.

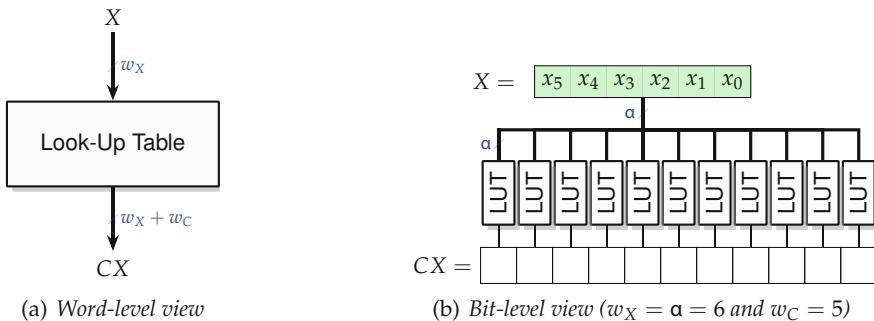


Fig. 12.9 Tabulating the product by C of all the possible values of X .

This is even relevant in practice for small values of w_X . Specifically, if $w_X \leq \alpha$, then each output bit of the product CX will cost exactly one FPGA LUT, as Fig. 12.9b shows.

However, this approach scales poorly as w_X increases. For $w_X = \alpha + 1$, each output bit costs 2 LUTs, plus one address-decoding multiplexer (which may be available for free in the FPGA fabric, or not). In general, for $w_X = \alpha + k$ -bit inputs, each output bit costs 2^k LUTs. The following presents a table-based technique that scales better with w_X .

12.2.2 The Table-and-Addition KCM Algorithm

The Ken Chapman's Multiplier (KCM) was introduced by Chapman [Cha94] and further studied by Wirthlin [Wir04]. It is an efficient way to use the look-up tables (LUTs) that form the logic fabric of the target FPGA.

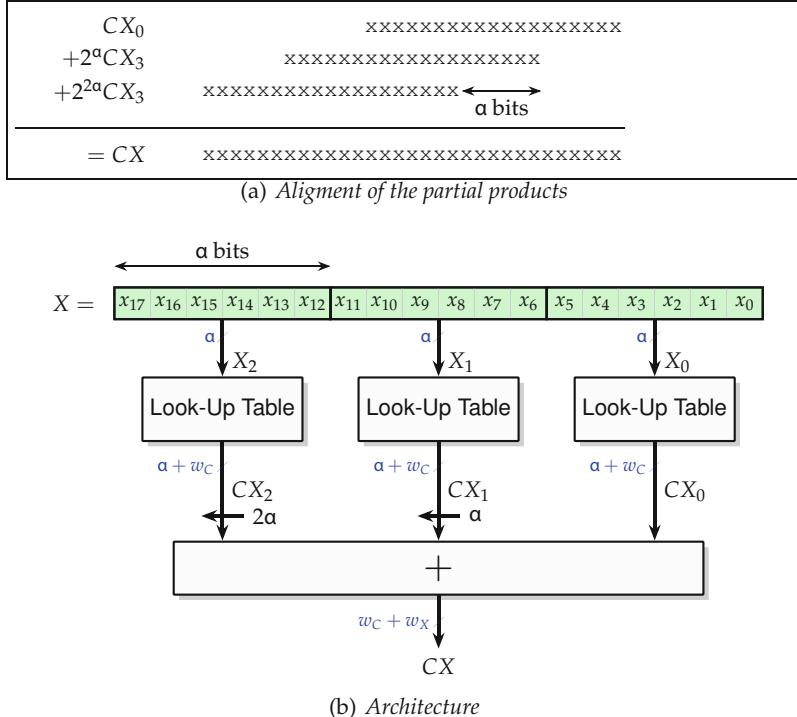


Fig. 12.10 KCM multiplication of a 18-bit constant C by an 18-bit input X split in 3 chunks of $\alpha = 6$ bits. Each subproduct CX_i is of size 24 bits.

The KCM technique is based on a high-radix representation (see Sect. 2.3) of the input X : As illustrated by Fig. 12.10, the binary representation of X is decomposed into consecutive chunks of α bits (for instance, hexadecimal is radix 16 for a decomposition in 4-bit chunks, and Fig. 12.10 shows a radix-64 decomposition). Mathematically,

$$X = \sum_{i=0}^{\lceil \frac{w_X}{\alpha} \rceil - 1} X_i \cdot (2^\alpha)^i, \text{ where } X_i \in \{0, \dots, 2^\alpha - 1\}. \quad (12.13)$$

From (12.13) the product of X by C can be written

$$CX = \sum_{i=0}^{\lceil \frac{w_X}{\alpha} \rceil} CX_i \cdot 2^{\alpha i} \quad (12.14)$$

which is illustrated in Fig. 12.10. We have a sum of (shifted) partial products CX_i , each of which is a $w_C + \alpha$ -bit integer. The KCM algorithm consists in reading each CX_i from a table of precomputed values, indexed by X_i , before summing them. Again, the cost of each table is one FPGA LUT per output bit.

The simplest way of computing the sum is to use a rake of $\lceil \frac{w_X}{\alpha} \rceil - 1$ adders in sequence. Due to the shifts, the lower α bits of each sum can be output directly. Therefore, each adder is of size w_C bits.

The total size of the KCM architecture is then

$$\#LUTs = \lceil \frac{w_X}{\alpha} \rceil \cdot (w_C + \alpha) + (\lceil \frac{w_X}{\alpha} \rceil - 1) \cdot w_C$$

This area is always very predictable and, contrary to the shift-and-add methods, almost independent on the value of the constant (although logic optimizers will still attempt to compress the tables and sometimes succeed).

If the input is large (therefore split in many chunks), an adder tree, or a compressor tree as discussed in Chap. 7, will have a shorter latency and potentially improve area.

There are many possible variations on the KCM idea:

- As all the tables contain the same data, a sequential version can be designed.
 - This algorithm is easy to adapt to signed numbers in two's complement.
 - Wirthlin [Wir04] studied the following optimization on Xilinx FPGAs: If we split the input in chunks of $\alpha - 1$ bits, then one row of LUTs can integrate both the table and the corresponding adder and still exploit the fast carry logic of Xilinx circuits. This reduces the overall area. This is one instance of the LUT+addition optimization opportunity discussed in Sect. 5.4.
- Intel FPGAs do not need this trick, since their elementary logic unit includes a LUT coupled with a full adder.
- Section 12.3.3 will show how it can be adapted to the product of a fixed-point input by an arbitrary real constant (like π or $\log 2$) with last-bit accuracy.

12.3 Multiplication of a Fixed-Point Number by a Real Constant

This section considers the problem of multiplying by an arbitrary real constant, such as π or $\log(2)$. Many applications involve multiplications by real constants. The implementation of a floating-point exponential in Chap. 22 and of sine and cosine in Chap. 20 will require products by $\log(2)$ and

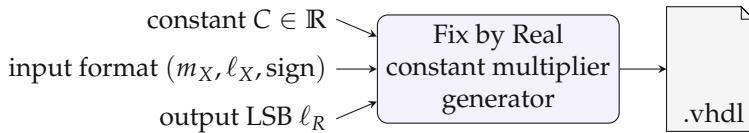


Fig. 12.11 Interface to a generator of multipliers by real constants.

$1/\log(2)$ and π and $1/\pi$, respectively. Many signal-processing filters and transforms are defined in terms of multiplications by real constants. The most pervasive is probably the FFT, whose twiddle factors are complex multiplications by $e^{\frac{k\pi}{n}}$, implemented using multiplications by $\sin \frac{k\pi}{n}$ and $\cos \frac{k\pi}{n}$. We can also mention standard filters whose (constant) coefficients are specified as mathematical formulas. This is the case, for instance, of pulse-shaping FIR filters (half-sine and root-raised cosine), as used in the ZigBee standard [802.15.4] among others.

Note that although we consider a real constant C , the input X is still a fixed-size number. As per Chap. 2, we consider here a fixed-point number denoted as $\text{ufix}(m_X, \ell_X)$ or $\text{sfix}(m_X, \ell_X)$, where m_X and ℓ_X are the binary weights of the MSB and LSB of X , respectively. Integers are a special case when $\ell_X = 0$. Multiplication of a floating-point number by a real constant will be addressed in Chap. 15, Sect. 15.1.

The interface to such a generator is shown in Fig. 12.11. Remark that we now need to provide the output LSB ℓ_R . Indeed, since a real constant may have an infinite number of bits, we need to specify to which precision the output product must be rounded.

The general idea is to round the constant C to some precision and then use one of the constant multiplication techniques presented above. This section shows specific refinements of this idea in the case of shift-and-add techniques (in Sect. 12.3.2) and table-based techniques (in Sect. 12.3.3).

12.3.1 Tabulated Perfectly Rounded Constant Multipliers

As in Sect. 12.2.1, the simplest option, for small input sizes, is to tabulate the product (Figs. 12.12, and 12.9b for a bit-level view). Here the value tabulated in the entry number i is $\lfloor C \cdot i \rceil_{\ell_R}$, the perfect rounding to the nearest of the product $C \cdot i$, which may be computed using arbitrary-precision software.

This technique involves one single rounding error, whose maximum value is one half-unit in the last place (ulp), or $2^{-\ell_R-1}$. It is the best accuracy one may achieve.

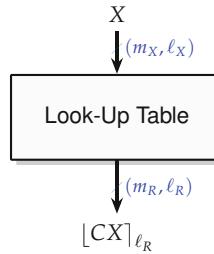


Fig. 12.12 Tabulating the perfectly rounded product by C of all the possible values of X .

Hands on: Tabulated constant multipliers in FloPoCo

The FloPoCo implementation of Fix-by-Real KCM presented in Sect. 12.3.1 defaults to tabulation for very small input sizes.

It is also possible to use the universal function generator introduced in Sect. 16.4 with the function $f(x) = Cx$ as argument. The following command produces an 8-bit in, 24-bit out multiplier by $2/\pi$:

```
flopoco FixFunctionByTable f="2/pi*x" \
signedIn=false lsbIn=-8 lsbOut=-24
```

The rest of this section addresses the construction of faithful architectures that scale better to larger input sizes.

12.3.2 Faithful Fix-by-Real Using Shift-and-Add

The construction of faithful shift-and-add fix-by-real constant multipliers is a recent topic [Din+19; GVK22; GV23].

As a shift-and-add graph can only be built for a finite-precision constant, one aspect of the problem is to find some finite-precision fixed-point \tilde{C} that suitably approximates the constant C (Fig. 12.13).

Assuming \tilde{C} is chosen, the error of the operator is defined as

$$\delta_{\text{total}}(X) = R(X) - CX \quad (12.15)$$

where $R(X)$ is the output of the operator when inputting X . The shift-and-add graphs of previous sections compute the exact multiplications: $P(X) = \tilde{C}X$. Usually the product $P(X)$ will have bits to the right of the output LSB ($\ell_P < \ell_R$); therefore, it needs to be rounded to obtain the output $R(X)$. The error $\delta_{\text{total}}(X)$ is therefore decomposed as

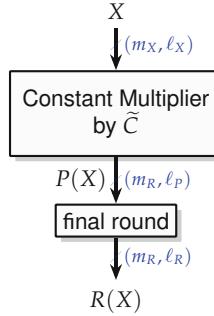


Fig. 12.13 Generic architecture for a faithful constant multiplier.

$$\delta_{\text{total}}(X) = R(X) - P(X) + P(X) - CX \quad (12.16)$$

$$= \delta_{\text{final round}} + \tilde{C}X - CX \quad (12.17)$$

$$= \delta_{\text{final round}} + (\tilde{C} - C)X \quad (12.18)$$

$$= \delta_{\text{final round}} + \delta_C X, \quad (12.19)$$

where $\delta_{\text{final round}}$ is bounded by 2^{ℓ_R-1} and δ_C is the error of rounding C to \tilde{C} . To achieve last-bit accuracy, $\delta_{\text{total}}(X)$ should always be smaller than 2^{ℓ_R} , which requires that

$$\forall X \quad |\delta_C X| < 2^{\ell_R-1}. \quad (12.20)$$

As $|X|$ is bounded by some X_{\max} (which depends on the format, in particular the signedness), a faithful multiplier requires that

$$|\delta_C| < 2^{\ell_R-1} / X_{\max}. \quad (12.21)$$

This defines the bound on δ_C that ensures faithful rounding:

$$\bar{\delta}_C = \frac{2^{\ell_R-1}}{X_{\max}}. \quad (12.22)$$

To reach this bound, one option is to consider the rounding $\lfloor C \rfloor_\ell$ of C to the nearest number of LSB ℓ . It defines a family of arbitrarily accurate approximations to C , since by definition, $\lfloor C \rfloor_\ell$ is such that

$$|\lfloor C \rfloor_\ell - C| \leq 2^{\ell-1}. \quad (12.23)$$

There exists an ℓ_{\max} such that $\forall \ell \leq \ell_{\max}, |\lfloor C \rfloor_\ell - C| < \bar{\delta}_C$. The fixed-point constant $\lfloor C \rfloor_{\ell_{\max}}$ has the minimum word size among the \tilde{C} fulfilling the faithful rounding condition. A shift-and-add graph for $\lfloor C \rfloor_{\ell_{\max}}$ therefore provides a faithfully rounded multiplier by C .

However, (12.22) also defines the interval where we may look for \tilde{C} :

$$C - \bar{\delta}_C < \tilde{C} < C + \bar{\delta}_C, \quad (12.24)$$

and in this interval, there is an infinite number of other values of \tilde{C} , for LSBs $\ell \leq \ell_{\max}$, that also fulfill the faithful rounding condition. This is illustrated by Fig. 12.14.

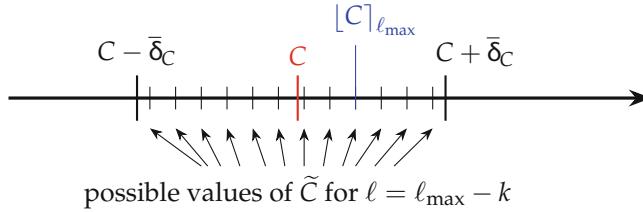


Fig. 12.14 Possible values of fixed-point coefficient \tilde{C} .

It is useful to search this interval for values of \tilde{C} which have a slightly larger word size than $[C]_{\ell_{\max}}$ (i. e., have an LSB $\ell = \ell_{\max} - k$ for a small positive integer k) for two reasons: Firstly, as the shift-and-add graph strongly depends on the constant, it is very likely [GQ10] that some of these wider constants will be cheaper to implement than $[C]_{\ell_{\max}}$. Secondly, a smaller δ_C may allow for truncations in the shift-and-add circuit [Din+19].

Indeed, it is possible to modify the ILP model presented in Sect. 12.1.4 to capture the truncation of some of the shifted values in a shift-and-add graph. It requires to model the corresponding truncation errors, but also the cost saving as the corresponding full adders are removed from the adder graph. It is also interesting to attempt to integrate the final rounding half-ulp bit in one of the adders of the adder graph. The objective function may thus be modified to maximize the number of full adders that will be removed, while keeping the sum of truncation errors below the faithful rounding bound. The interested reader will find more details in [Din+19] and an improved ILP model of the truncated constant multiplication in [GVK22; GV23].

There is also an ℓ'_{\max} such that rounding to nearest the product $[C]_{\ell'_{\max}} X$ always provides the *correct* rounding of CX to the nearest. This issue is briefly surveyed in Sect. 15.1.4. Typically, $\ell'_{\max} \approx \ell_{\max} - w_X$ where w_X is the word size of X . This is very expensive for one more bit of accuracy, and it is difficult to find a use case that needs it.

12.3.3 Faithful Fix-by-Real KCM Algorithm

The original KCM method, presented in Sect. 12.2, addresses the multiplication by an integer constant and computes an exact product. We here present a fix-by-real KCM technique that performs the multiplication by a *real* constant and is therefore necessarily approximate. It also generalizes KCM to a fixed-point format (m, ℓ) .

This method still consists in breaking down the binary representation of input X into D chunks d_k , each of α bits. With the input size being $m_X - \ell_X + 1$, we have

$$D = \lceil (m_X - \ell_X + 1) / \alpha \rceil. \quad (12.25)$$

Mathematically, we first scale X to an integer $2^{-\ell_X} X$, which is decomposed as shown in Fig. 12.15:

$$2^{-\ell_X} X = \sum_{k=0}^{D-1} 2^{k\alpha} X_k \quad \text{where} \quad X_k \in \{0, \dots, 2^\alpha - 1\}. \quad (12.26)$$

Again, another point of view is that the scaled input $2^{-\ell_X} X$ is considered as a radix- 2^α number, the X_k s being its digits—think hexadecimal when $\alpha = 4$.

The product becomes

$$CX = \sum_{k=0}^{D-1} 2^{k\alpha + \ell_X} CX_k. \quad (12.27)$$

Since each chunk X_k consists of α bits, where α is the LUT input size, we may tabulate each product CX_k in a lookup table with α input bits. Let us now address its number of output bits.

CX_k has an infinite number of bits in the general case (think $C = \pi$), so it must be rounded. We will round it to precision $\ell_R - g$, where ℓ_R is the precision we want for the result, and g is a small positive integer, a number of *guard bits* that will be determined by the error analysis to ensure that the result is faithful to precision ℓ_R .

Let us call $T_k(X_k) = \left\lfloor 2^{k\alpha + \ell_X} CX_k \right\rfloor_{\ell_R-g}$ this rounded value. Figure 12.15a describes the alignment of the $T_k(X_k)$ in a KCM architecture.

Note that contrary to classic (integer) KCM, each tables does not consume the same amount of resources. The factor $2^{k\alpha}$ in (12.27) shifts the MSB of the table output T_k , as illustrated by Fig. 12.15a.

A standard KCM invoked with some constant $\approx C$ (in other words using a KCM constant multiplier in the generic architecture of Fig. 12.13) would lead to a different alignment, more similar to Fig. 12.10. It would be wasteful, as some tables would store bits that do not really participate to the final result.

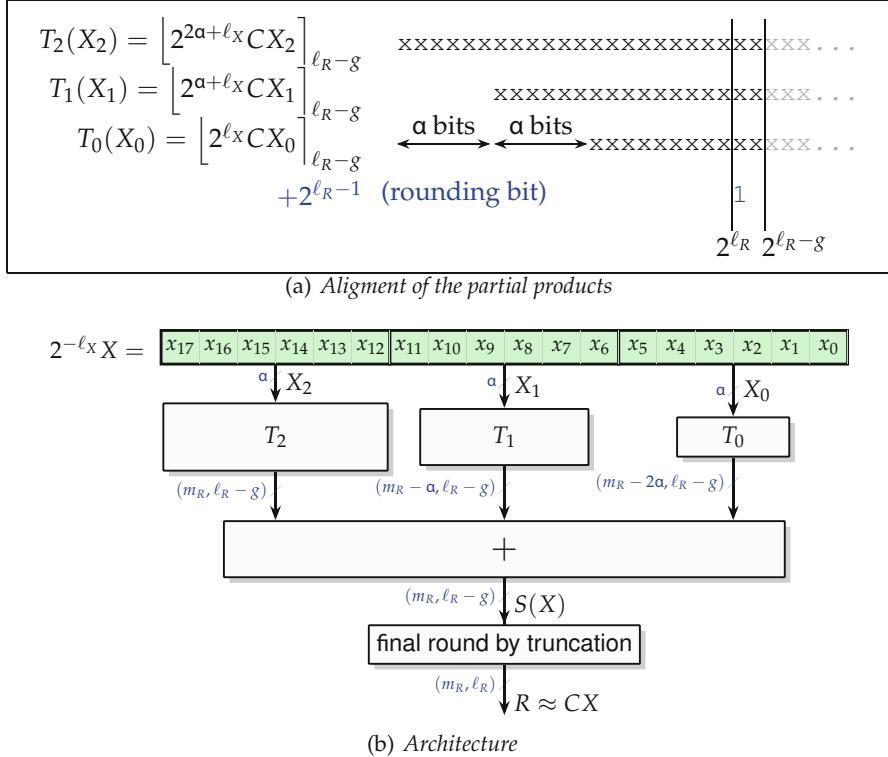


Fig. 12.15 Fix-by-real KCM. The 18-bit input X is split in $D = 3$ chunks of $a = 6$ bits.

12.3.3.1 Error Analysis

The total absolute error of a FixRealKCM multiplier architecture is defined as the difference between the computed value and the ideal value:

$$\delta_{\text{total}}(X) = R(X) - CX. \quad (12.28)$$

Last-bit accuracy means that this error should be smaller than the ulp of the output: The objective is to achieve

$$\bar{\delta}_{\text{total}} < 2^{\ell_R}. \quad (12.29)$$

The fixed-point additions are errorless (and the value of m_R is chosen to ensure that no overflow may occur). The error $\delta_{\text{total}}(X)$ therefore only has two sources: the final rounding and the rounding performed when filling each table. Let us call $S(X)$ the intermediate sum before the final rounding (see Fig. 12.15):

$$S(X) = \sum_{k=0}^{D-1} T_k(X_k). \quad (12.30)$$

Then the previous error decomposition is expressed mathematically as

$$\begin{aligned} \delta_{\text{total}}(X) &= R(X) - CX \\ &= \underbrace{R(X) - S(X)}_{=\delta_{\text{final round}}(X)} + \underbrace{S(X) - CX}_{=\delta_{\text{round}}}. \end{aligned} \quad (12.31)$$

The error term $\delta_{\text{round}}(X)$ is the sum of the rounding errors committed when filling each table:

$$\begin{aligned} \delta_{\text{round}}(X) &= S(X) - CX \\ &= \sum_{k=0}^{D-1} T_k(X_k) - C \sum_k 2^{\ell_X+k\alpha} X_k \quad \text{from (12.27)} \\ &= \sum_{k=0}^{D-1} (\left[2^{k\alpha+\ell_X} CX_k \right]_{\ell_R-g} - 2^{\ell_X+k\alpha} CX_k). \end{aligned} \quad (12.32)$$

By definition of the rounding at precision $\ell_R - g$, we obtain that

$$|\delta_{\text{round}}(X)| < D \times 2^{\ell_R-g-1} = \overline{\delta_{\text{round}}}. \quad (12.33)$$

This error bound is proportional to 2^{-g} , so it can be made as small as needed by increasing g .

The final sum must be rounded to the target precision ℓ_R . This rounding to the nearest can be implemented as a simple truncation, provided that the value of the rounding bit 2^{ℓ_R-1} has been added (for free) to all entries of one of the tables (T_0 in Fig. 12.15a, where the rounding bit is in blue). The final rounding error $\delta_{\text{final round}}(X)$ is bounded by

$$\delta_{\text{final round}}(X) \leq \overline{\delta}_{\text{final round}} = 2^{\ell_R-1}. \quad (12.34)$$

Finally, we obtain from (12.31), (12.33), and (12.34) the total error bound as a function of g :

$$\overline{\delta}_{\text{total}} = 2^{\ell_R-1} + D \times 2^{\ell_R-g-1} \quad (12.35)$$

The cost of the architecture obviously grows with g , which we therefore want to minimize. The smallest value of g that allows for a faithful (or last-bit accurate) operator, i. e., $\overline{\delta}_{\text{total}} < 2^{\ell_R}$ as defined in Chap. 3, is therefore

$$g > \log_2 D. \quad (12.36)$$

Note that half-unit biased (HUB) rounding could also be used, improving the final rounding error to

$$\delta_{\text{final round}}(X) \leq \bar{\delta}_{\text{final round}} = 2^{\ell_R - 1} (1 - 2^{-g}) \quad (12.37)$$

hence,

$$g > \log_2(D - 1). \quad (12.38)$$

However, this is currently not implemented in FloPoCo.

12.3.3.2 Implementation Considerations

It is often possible in practice to use a finer bound for δ_{round} than the D half-ulps of (12.33). To start with, some constant multipliers entail no error: it is, for instance, the case for multiplication by 0 and by 1. Such trivial cases will happen surprisingly often when the FixRealKCM generator is used as a backend for a larger architecture generator, in particular for finite impulse response (FIR) or infinite impulse response (IIR) filters [Vol+19]. Besides, such trivial cases deserve specific treatment, since their implementation is much simpler than the generic case and may save resources.

There are many other useful cases when the results of several constant multiplications are summed in a coarser operator. Another significant example is the multiplication of a complex input $X_{\text{re}} + iX_{\text{im}}$ by a complex constant $C_{\text{re}} + iC_{\text{im}}$, which can be performed by computing

$$R_{\text{re}} = C_{\text{re}}X_{\text{re}} - C_{\text{im}}X_{\text{im}} \quad (12.39)$$

$$R_{\text{im}} = C_{\text{re}}X_{\text{im}} + C_{\text{im}}X_{\text{re}} \quad (12.40)$$

The remainder of this Sect. 12.3.3.2 describes how specific constant values may be managed in a way that enables a global error analysis in such contexts.

Constant-Specific Error Bounding

Here is the list of cases currently managed by the FloPoCo FixRealKCM implementation. The rounding error bounds $\overline{\delta}_{\text{round}}$ below are expressed in *gulp*, where the gulp value is the ulp of the computation including the guard bits (its value is $2^{\ell_R - g}$). Using this unit will allow each constant multiplier to evaluate and report its error before knowing the value of g , which will prove useful in the sequel.

1. $C = 0$: then $\overline{\delta}_{\text{round}}^{(\text{gulp})} = 0$ and the implementation of the multiplier is trivial.
2. $|C| = 2^n$: we want to replace this multiplier by a constant shift, costing only wires. Then $\overline{\delta}_{\text{round}}^{(\text{gulp})} = 0$ if $n + \ell_X \geq \ell_R$ (shift of X such that all the bits will be kept); otherwise, $\overline{\delta}_{\text{round}}^{(\text{gulp})} = 1$ (shift to the right, losing some bits)

due to truncation). Here we may overestimate the error, because the test should be if $n + \ell_X \geq \ell_R - g$, but we do not know g yet.

3. C is an integer multiple of $2^{\ell_R - \ell_X}$: as X is either 0, or an integer multiple of 2^{ℓ_X} , then for $X > 0$, $|CX| > 2^{\ell_R}$. In other words the multiplication produces no bit to the right of 2^{ℓ_R} . Therefore, whatever $g \geq 0$, no rounding will be needed; hence, $\delta_{\text{round}}^{(\text{gulp})} = 0$ (since the case $X = 0$, of course, also entails $\delta_{\text{round}}^{(\text{gulp})} = 0$). This analysis is actually better performed on a per-table basis, as illustrated by Fig. 12.16.
 4. In the general case, $\delta_{\text{round}}^{(\text{gulp})} = D/2$ as per (12.33) (we have D tables, each entailing one half-up of error).

One final technicality: We have so far assumed that the number of tables D is computed out of the input size, using (12.25). However, for small constants, it may happen that the contribution of the lower tables can be neglected. To understand this, consider Fig. 12.15a: Each table output is shifted right if C_i is small. Therefore, the generator will drop a table as soon as its MSB is smaller than $\ell_R - g - 1$. The error analysis remains valid in this case, although the source of the error is no longer the rounding of the table, but it is being neglected altogether. If more than one table is fully neglected, this error analysis was slightly pessimistic (we could have a single half-ulp for all the neglected tables), but it remains safe.

Fig. 12.16 Specific case of Fig. 12.15a when C fits on few bits (here 6). Each table output is now exactly an $\alpha + 6 = 12$ -bit number and needs to be rounded only when some of its bits have their weights smaller than $\ell_R - g$ (in this example, only T_0). The other tables are exact and therefore do not contribute to the error. In this example the overall rounding error due to the tables is only $\overline{\delta_{\text{round}}} = 2^{\ell_R - g - 1}$ instead of $3 \times 2^{\ell_R - g - 1}$ as per (12.33). This in turns allows for a smaller value of g . Finally, note that the height of the resulting bit heap is only 2.

Bit Heaps and Virtual KCM Operators

The summation shown in Fig. 12.15 can be implemented by a compressor tree using the bit heap framework. This is particularly relevant when summing the results of several KCMs, in which case one single compressor tree can be generated.

Therefore, it is important to provide operators such as `FixRealKCM` as *virtual* operator generators: They input a bit heap, generate hardware that compute some bits, add these bits to the bit heap, but do not compress the bit heap—compression is delegated to the enclosing operator (e. g., the complex constant multiplier).

Just as it is more efficient to perform a global optimization of the whole summation, it is more efficient to perform a single global error analysis. To understand it, consider how we would compute $R = C_1X + C_2Y$ in a naive way:

- To achieve last-bit accuracy for R , we would need to compute the sum with $g = 2$ guard bits.
- Each KCM would therefore be built to be last-bit accurate to $2^{\ell_R - 2}$.
- For this purpose, each KCM for C_i would compute internally that it needs g_i guard bits. Assume, for instance, that $D = 3$; hence, $g_i = 2$.
- Tables would be computed and summed with an LSB of $\ell_R - 4$, with a final rounding of each KCM to $2^{\ell_R - 2}$.

What is wrong in this picture is the two intermediate roundings. Indeed, we could achieve last-bit accuracy with all the six tables rounded to $2^{\ell_R - 3}$.

A generic algorithm that manages special constant cases within a shared bit heap is the following:

- In a first pass, evaluate the $\overline{\delta_{\text{round}}^{(\text{gulp})}}$ for each constant.
- Sum all these errors.
- Compute the value of g that will enable last-bit accuracy, assuming a single final round to ℓ_R .
- In a second pass, actually generate the hardware for this value of g .

A detailed example of such algorithm for a sum of products by constants (SOPC) will be given in Sect. 12.5.5.

12.4 Multiplication by a Rational Constant

This section presents an algorithm that builds a multiplier by a rational constant A/B , based on its periodic binary representation. The study of periodic representations has very old decimal roots [Gla78], and we extend to arbitrary rational constants early works related to division by a constant, mostly in a software context [AHS76; Li85]. The interested reader will find a unifying survey on constant division in [SP94], including several other references specific to division by 10 needed for binary/decimal conversions. The generalization to arbitrary rational constants and the application to hardware is a systematic exploitation of an empirical observation made by Gustafsson and Qureshi [GQ10].

Section 12.4.1 shows how to compute the periodic binary representation of an arbitrary rational A/B . Section 12.4.2 shows how to construct optimal adder graphs out of it. Section 12.4.3 shows that a KCM approach also benefits from the periodicity of the constant.

12.4.1 On the Periodicity of the Binary Representation of Rational Numbers

This section is based on number representation in the classic position system, where an integer is represented by a sequence of digits, and each digit is weighted by powers of some radix, usually 10 (decimal system) or 2 (binary system).

In such a system, any rational number A/B has an eventually periodic representation. This is true in decimal ($1/3 = 0.3333 \dots$, $1/9 = 0.1111 \dots$) but also in binary ($1/3 = 0.01010101_2 \dots$, $1/9 = 0.000111000111000_2 \dots$). Numbers with a finite decimal representation can be viewed as a special case where the periodic pattern is composed of zeroes, for instance, $0.5 = 0.50000 \dots$. A number with a finite decimal representation may have an infinite binary one, for instance, $1/5 = 0.2_{10} = 0.001100110011 \dots$. The opposite is not true, due to the fact that two divides ten.

The following lemma tells us which numbers have a purely periodic binary representation:

Lemma 12.1 *Let us consider an irreducible fraction C/D , where 2 divides neither C nor D . If $C < D$, then the binary representation of C/D is purely periodic, i.e., it starts with an occurrence of the periodic pattern.*

The condition that 2 divides neither C nor D is not a constraint, since powers of two correspond to shifts in binary. For most purposes, they may be handled separately in a trivial way.

If $C > D$ the Euclidean division of C by D gives us $C = HD + C'$, and we may rewrite $C/D = H + C'/D$. For the purpose of multiplying an input X by the constant C/D , we therefore have $XC/D = XH + XC'/D$. Previous sections of this chapter address the multiplication by the finite integer constant H , so we may focus on the multiplication by the purely periodic constant C'/D .

The following lemma shows how to compute the periodic pattern:

Lemma 12.2 *Let C/D be an irreducible fraction, where $C < D$, and 2 divides neither C nor D . The size s of its period is the multiplicative order of 2 modulo D , i.e., the smallest integer such that $2^s \bmod D = 1$. The periodic pattern is the integer $P = \lfloor 2^s C/D \rfloor$.*

Proof By definition of s we have $2^s = kD + 1$ for some integer k . Therefore, $P = \lfloor 2^s C/D \rfloor = \left\lfloor \frac{(kD+1)C}{D} \right\rfloor = \lfloor kC + C/D \rfloor = kC$ since $C/D < 1$. We deduce that

$$2^s C/D = P + C/D,$$

where the recursive occurrence of C/D exactly expresses the periodicity of the fraction C/D .

Examples:

- $1/3$ has period size $s = 2$ because $2^2 \bmod 3 = 1$. The pattern is $\lfloor 1 \times 2^2/3 \rfloor = 1$, which we write on 2 bits 01, and we obtain that

$$1/3 = 0.(01_2)^\infty.$$

- $5/9$ has period size $s = 6$ because $2^6 \bmod 9 = 1$. The pattern is $\lfloor 5 \times 2^6/9 \rfloor = 35$, which we write on 6 bits 100011, and we obtain that

$$5/9 = 0.(100011_2)^\infty.$$

Thanks to these lemmas, algorithm 12.2 determines the periodic binary representation of a fractional number A/B . This representation consists of four integers:

- s the period size in bits, a positive integer,
- P the periodic pattern, a positive integer that we will usually write in binary,
- H the header, a positive integer, also typically written in binary,
- e the scaling factor exponent, or shift,

such as

$$A/B = 2^e \left(H + \sum_{i=1}^{+\infty} \frac{P}{2^{si}} \right).$$

Let us now exploit this representation to build a multiplier of a variable X by a fraction A/B .

12.4.2 Periodical Shift-and-Add Graphs

In this section, the input is the periodic representation of a rational constant A/B and also a precision w_0 , which is the number of bits of A/B that have to be considered for the multiplication. The value of w_0 typically expresses the accuracy requirements of the floating-point or fixed-point context. For instance, Sect. 15.1.5 will define the value of w_0 that ensures correct rounding for a given floating-point format.

Algorithm 12.2: Computing the periodic representation of a rational A/B as a tuple of integers (e, H, P, s)

```

function PeriodicBinaryRepresentation( $A, B$ )
     $(C, D) \leftarrow \text{Simplify}(A, B)$       //  $C/D = A/B$  and  $C/D$  irreducible
    // Now remove the factor 2 from both numerator and denominator
     $e \leftarrow 0$ 
    while  $C \bmod 2 = 0$  do
         $C \leftarrow C/2$ 
         $e \leftarrow e + 1$ 
    end while
    while  $D \bmod 2 = 0$  do
         $D \leftarrow D/2$ 
         $e \leftarrow e - 1$ 
    end while
    // Now  $A/B = 2^e C/D$  with both  $C$  and  $D$  odd
     $(H, F) \leftarrow \text{EuclideanDiv}(C, D)$       //  $H$ : quotient,  $F$ : remainder
    // Now  $A/B = 2^e(H + F/D)$  with  $F < D$ 
     $s \leftarrow 1$                                 //  $s$  will be the period size
    if  $D = 1$  then
         $P \leftarrow 0$                           // finite binary representation, no periodic pattern
    else
        //  $A/B$  has an infinite binary representation
         $t \leftarrow 2$                           // Invariant of the loop below:  $t = 2^s$ 
        while  $t \bmod D \neq 1$  do
             $t \leftarrow 2t$ 
             $s \leftarrow s + 1$ 
        end while
         $P \leftarrow Ct/D$                       // periodic pattern
    end if
    return  $(e, H, P, s)$ 

```

In [GQ10], Gustafsson and Qureshi suggested trying to represent a real constant on more than w_0 bits if it leads to a shift-and-add architecture with fewer additions. They indeed mention the fact that, due to their periodic representation, rational constants are good candidates for exploiting this idea, without exploiting this idea systematically.

With the notations of previous section, let us define

$$\pi_0 = 2^{-s} PX.$$

The 2^{-s} factor simply scales the integer P to an approximation of $C/D < 1$, so π_0 is an approximation of CX/D . We may then compute increasingly accurate approximations of CX/D as

$$\pi_1 = \pi_0 + 2^{-s} \pi_0,$$

$$\pi_2 = \pi_1 + 2^{-2s} \pi_1,$$

and in general

$$\pi_{i+1} = \pi_i + 2^{-2^i s} \pi_i$$

so that π_i is the product of X by an approximation of C/D of size $2^i s$ bits.

Therefore, a constant corresponding to 2^i repetitions of the period may be built in i additions, and this is optimal [Gus07b].

Let us now consider the details, including further optimizations. We note w_H the size in bits of the header H . We need to compute $HX + CX/D$, where CX/D is the periodic part. If w_0 is the precision to which $H + C/D$ must be represented, then C/D must be represented at least on $w_0 - w_H$ bits.

First, one of the existing methods is used to build PX and, if $H \neq 0$, HX . As these two constants should be small for the method to be relevant, exhaustive exploration techniques [DM94; Gus+06; TN11] may be used to perform this step optimally.

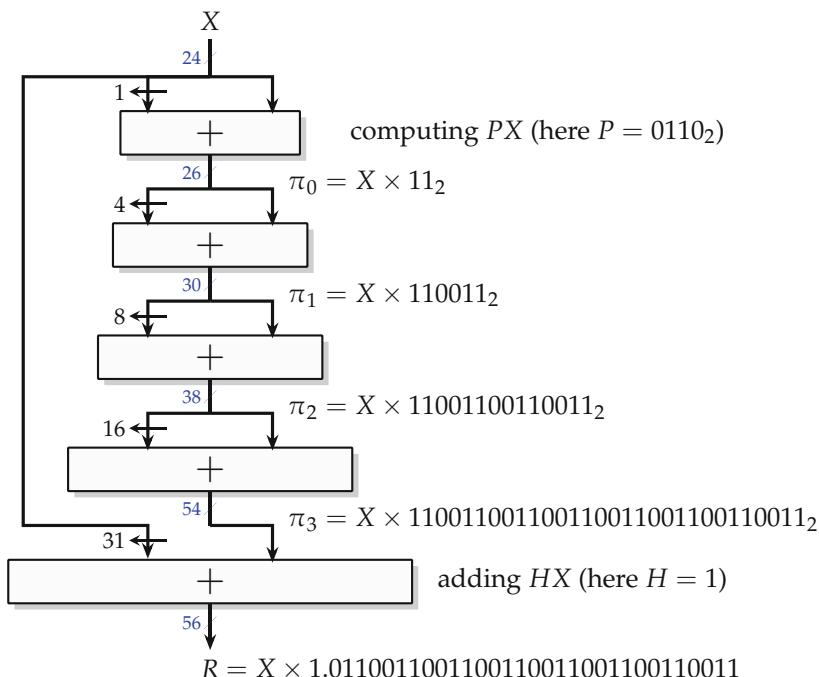


Fig. 12.17 Multiplication of a 24-bit mantissa by 7/5 (period $P = 0110_2$, header $H = 1$) for a result correctly rounded to 2^{-24} . The size of the adders in the figure is proportional to their cost (in full adders).

Then we may compute the π_i . In this process, we may stop as soon as $2^i s \geq w_0 - w_H$. However, it is usually possible to implement a smaller last addition. Let i be such that $2^i s < w_0 \leq 2^{i+1}s$: We must compute the π_j for $0 \leq j \leq i$. Let j be the smallest integer such that $(2^i + 2^j)s \geq w_0 - w_H$. As $j \leq i$, π_j is already computed, and the last stage may compute

$$R = \pi_j + 2^{-2^i s} \pi_i$$

(another option would be to compute $R = \pi_i + 2^{-2^j s} \pi_j$, but this would lead to a larger adder, see Sect. 12.1.6).

If $H = 0$, this is all. If $H \neq 0$, we still have to add HX . This product is itself computed using a constant multiplier, in parallel to the computation of the fractional product. There are two possible parenthesizing of the two final additions: $R = (HX + \pi_j) + 2^{-2^i s} \pi_i$ or $R = HX + (\pi_j + 2^{-2^i s} \pi_i)$. We may assume that the computation of HX has a depth strictly smaller than that of π_i (which should be the case for “small” rationals). With this assumption, as soon as $j < i$, the first parenthesizing leads to a shallower graph and is therefore preferred. If $i = j$, the second parenthesizing will be preferred when it leads to a smaller overall number of full adders.

Figure 12.17 illustrates the resulting architecture on the example of $A/B = 7/5$ for single precision, with a target precision $w_0 = 28$ (which ensures correct rounding for binary32 floating point, according to Sect. 15.1.5). The smallest value of i such that $2^i s \geq 28$ is $i = 3$. We do not find in this case a smaller j such that $(2^{i-1} + 2^j)s \geq 28$. For this simple example the product PX is computed in one addition only, while the product HX is computed in zero additions.

This figure also illustrates a small additional optimization: We trim, whenever possible, leading and trailing zeroes from the various sub-constants to minimize datapath width. For instance, for $A/B = 7/5$, the period is $P = 0110_2$, but π_0 is actually computed as $X \times 11_2$ and the two zeroes are added only when performing the shifts. The final result is actually one bit more accurate than it seems, since there is one more trailing zero to the truncated constant. These technical details are taken into account by the generator in FloPoCo.

Here are a few concluding remarks. The logarithmic complexity of the above method is optimal [Gus07b]. An optimal algorithm is interesting in its own right, in a field where little is known in terms of theoretical complexity [DIZ07].

In the frame of the present book, however, a few important remarks should be made:

- In practice, exploiting periodicity is only useful when the period is short, which happens mostly for small values of the denominator B .
- The optimal algorithms presented in Sects. 12.1.3 and 12.1.4 will find the optimal adder graph as well, once the rounding of the rational constant

0/3	0000000000000000000000000
1/3	000010101010101010101010
2/3	000101010101010101010101
3/3	0010000000000000000000000
4/3	001010101010101010101010
...	...
14/3	100101010101010101010101
15/3	1010000000000000000000000

Fig. 12.18 For rational constants (here 1/3), the KCM tables are periodic.

has been determined. Still, the algorithm based on the period of the binary representation is always faster, and it scales much better to large sizes. It is also interesting in its own right, as an example on how the underlying mathematical structure of a constant can be exploited.

- These methods can be used to divide by a constant (just multiply by the inverse, which is a rational number). However, to divide by very small integers such as 3 or 5, specific specializations of the division algorithm are presented in Chap. 13 and should also be considered. They typically have a lower area, but possibly a larger latency. A detailed comparison of techniques for division by a constant will be discussed in Chap. 13.

12.4.3 Table-Based Multiplication by Rational Constants

Figure 12.18 is a table holding $X_i/3$ for X_i on $\alpha = 4$ bits.

Since each row, having the same denominator, is eventually periodic with the same period, one may observe that the whole table is eventually periodic. This example 22-bit table can be implemented as 5 LUTs instead of 22. In general, for a constant $A/B < 1$ of period size s , the table for A/BX_i requires about $\alpha + s$ LUTs: only the most significand α bits are not periodic.

This optimization is discovered by synthesis tools; therefore, it does not need to be explicated in the architectural description. However, its benefit is limited since it only reduces the size of the tables, not the size of the KCM addition hardware.

12.5 Multiplication by Multiple Constants

The single constant multiplication (SCM), as discussed so far, can be generalized in different directions.

- The multiple constant multiplication (MCM) problem addresses the multiplication of one input by several constants. In the shift-and-add approach, there is an opportunity of sharing intermediate products between several constant products. It is studied in Sect. 12.5.1. Section 12.5.2 shows that there are similar sharing opportunities in KCM-based MCM.
- The constant matrix multiplication (CMM) is a further generalization where a constant matrix is multiplied by a vector of variables. It is studied in Sect. 12.5.4.
- The sum of products by constants (SOPC) problem computes the scalar product of a vector of variables by a vector of constants. It is a central operation in digital filters such as FIR or IIR, among others. Here there is an opportunity, in a KCM approach, to fuse the KCM additions and the final addition in a single bit heap. It is studied in Sect. 12.5.3 in the integer case and in Sect. 12.5.5 in the case of real constants.

12.5.1 Multiple Constant Multiplication Using Shift-and-Add

The multiple constant multiplication (MCM) operation, i. e., the multiplication of a variable by several constants is a frequent operation in many digital signal processing (DSP) systems. The most prominent applications are digital filters but it also appears in discrete transforms like, e. g., the FFT or the discrete cosine transform (DCT). Of course, an MCM operation can be implemented by just using several SCM operations. However, as in each shift-and-add SCM operation, several intermediate factors are used, and they can be shared to decrease the number of total adders.

Take, for example, an MCM operation which has to multiply with constants 43 and 19. Figure 12.19a,b show their optimal SCM solutions (in terms of 2-input adders). Taking these two solutions would require five adders. One can observe by inspection that the factor $3x$ appears in both circuits, so one could remove one of the corresponding adders by resource sharing. However, as illustrated by the optimal MCM solution shown in Fig. 12.19c, another adder can be saved by considering that factor 43 can be computed from 19 and 5 as $43 = 19 \cdot 2 + 5$.

The search for efficient algorithms to solve the MCM problem has been an active research area for the last two decades. Many different optimization methods have been proposed. The n-dimensional reduced adder graph (RAG-n) algorithm, introduced by Dempster and Macleod [DM95], was one of the leading MCM heuristics for many years. This situation changed significantly with the introduction of the H_{cub} algorithm by Voronenko and Püschel [VP07] and the introduction of the difference-based adder graph (DiffAG) algorithm by Gustafsson [Gus07a], both in 2007. Both state-of-the-art MCM heuristics are briefly introduced in the following.

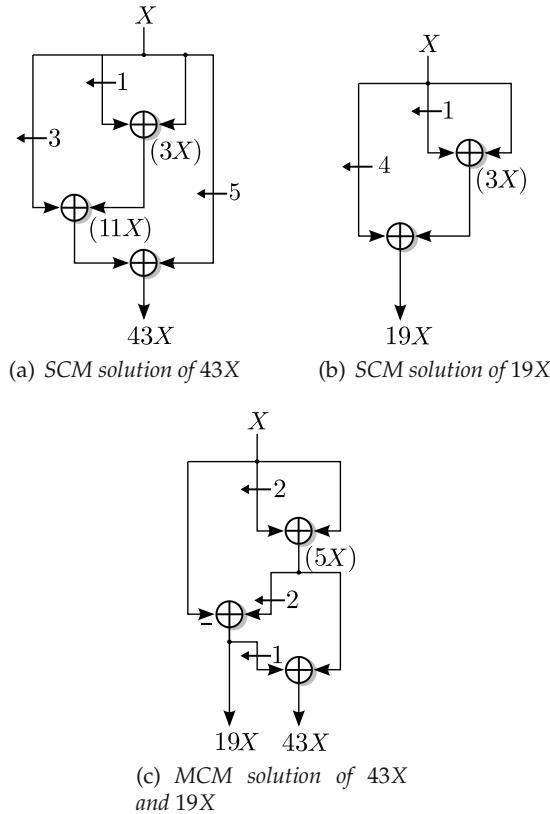


Fig. 12.19 Optimal adder circuits to realize a multiplication with coefficients 19, 43, and both 19 and 43.

The H_{cub} algorithm starts, like most other MCM algorithms, with an algorithm that is called the *optimal part* of the MCM optimization. In that part, all of the target coefficient set T that can be computed from '1' by using a single adder are moved into a so-called realized set R . Next, R is recursively updated with all coefficients that can be computed from R until no further target element can be realized. In case the remaining target set T is empty, it is known that the solution is optimal (each target can be computed by one adder). If elements in T remain, at least one additional non-output fundamental (NOF) from the successor set has to be inserted into R to compute the remaining elements of T .

For that, adder graph topologies with up to three adders are evaluated in the H_{cub} algorithm to obtain or estimate (in case that more than three additional adders are necessary) the so-called \mathcal{A} -distance of all possible successors in S . The \mathcal{A} -distance, denoted by $\text{dist}(R, c)$, is defined as the minimum number of \mathcal{A} -operations necessary to compute c from R . The main idea is to

select the successor s leading to the best benefit

$$B(R, s, t) = \text{dist}(R, t) - \text{dist}(R \cup \{s\}, t) \quad (12.41)$$

for all remaining target coefficients $t \in T$ (in fact, a slightly modified *weighted* benefit function is used [VP07]). The successor s with the best cumulative benefit (hence, the name cumulative benefit heuristic, H_{cub}) is included in R and the optimal part of the algorithm is evaluated again which probably moves further elements from T to R . This procedure is continued until T is empty. The implementation of the H_{cub} algorithm is available online as open source.¹

The DiffAG algorithm [Gus07a] follows a different strategy. It basically starts with the same optimal part as given above. Then, an undirected complete graph is formed where each node corresponds to a set of coefficients. Initially, one node corresponds to the realized set $N_0 = R$, and for each remaining target element, one node with a corresponding set containing one target element $N_i = \{t_i\}$ for all $t_i \in T \setminus R$ is created. Now, the so-called *differences* are computed between each pair of nodes in the graph and are inserted into the difference sets D_{ij} . A difference $d_{ij} \in D_{ij}$ is a fundamental that allows –when inserted into R – to compute the elements in N_i from the elements in N_j and vice versa. Each edge (i, j) corresponds to one set of differences D_{ij} . If there exists a difference which is included in the realized set ($d_{ij} \in R$), the corresponding nodes are merged into a single node $N_k = N_i \cup N_j$, N_i and N_j are removed and the corresponding edges, and their difference sets are merged. If no difference remains that is included in the realized set, the idea is to realize the least-cost and most frequent difference. This process is continued until all nodes in the graph are merged into a single node which set corresponds to the final solution. The results in [Gus07a] show that the DiffAG algorithm is advantageous for large target sets with low coefficient word size compared to the H_{cub} algorithm.

Solving the MCM problem in an optimal way received a lot of attention in the recent years. A first approach using integer linear programming (ILP) was proposed by Gustafsson [Gus08]. The search space of the MCM problem was constructed as a directed hypergraph where each node corresponds to a possible factor and hyperarcs (=two input arcs) are used to represent ways to compute this factor from other factors. Then, the MCM problem is identical to the problem of finding a Steiner hypertree in that hypergraph. However, its computational complexity is only suitable for small MCM instances or to find lower bounds of the adder count by relaxing the model to a continuous LP problem. Similar ideas were used in [Kum+13a; Kum15] to solve variants of the MCM problem: The pipelined MCM (PMCM) includes the minimization of pipeline registers (see Sect. 12.5.1.2); the minimum adder depth MCM problem is important for low-power applications. Another ILP-based approach was proposed by Aksoy et al. [Aks+08]. How-

¹ <http://www.spiral.net>.

ever, as it assumes a certain number representation, it is not globally optimal. The same group developed specialized breadth-first search (BFS) and depth-first search (DFS) algorithms for optimally solving the MCM problem [AGF08; AGF10].

12.5.1.1 ILP Extension for the MCM Problem

The ILP model for the SCM problem of Sect. 12.1.4 is now extended to the MCM problem [Kum18]. To consider multiple outputs, we have to add a new binary decision variable $o_{a,j}$ which determines the adder result a for each output $j = 1 \dots N_O$. This leads to the following constraints which replace $\mathcal{C}6$ in the MCM case

$$\mathcal{C}6': \quad C_j = c_a \text{ if } o_{a,j} = 1 \quad \forall a = 0 \dots N_A, j = 1 \dots N_O \quad (12.42)$$

where C_j denotes the j th target constant. Indicator constraints $\mathcal{C}6'$ ensure that for each constant, there exists one adder realizing the corresponding factor. They can be linearized as follows:

$$\mathcal{C}6'': \quad c_a - M + Mo_{a,j} \leq C_j \leq c_a + M - Mo_{a,j},$$

with $M > \max(C_i)$.

The lower bound of the adders can now be increased by the fact that we need for a total number of N_{uq} positive, odd and unique coefficients at least N_{uq} adders. This can be refined by adding the lowest of the SCM cost as defined in (12.43) of each of the coefficients, leading to [Gus07b]:

$$N_{A,\text{lb},\text{MCM}} = N_{\text{uq}} + \min_{t \in T} (\lceil \log_2(\text{nz}(t)) \rceil) - 1 \quad (12.43)$$

Refinements are possible which may slightly increase this lower bound for some coefficient sets. The interested reader may check [Gus07b].

Further extensions to address the bit-level cost as addressed in Sect. 12.1.5 or the multiplication with real constants as discussed in Sect. 12.3.2 can be found in [GVK22; GV23].

12.5.1.2 Pipelined Multiple Constant Multiplication

In case the performance of a shift-and-add SCM or MCM circuit is not sufficient, it may be pipelined [Mey+06] (an introduction to pipelining can be found in Sect. A.2.5). Pipeline registers may then contribute significantly to the cost: Finding the optimal pipeline schedule for a given adder graph [KZ11] is a new optimization problem, called the pipelined MCM (PMCM) problem. One heuristic to solve the PMCM problem is the reduced pipelined adder graph (RPAG) algorithm [Kum+12; Kum15]. It directly produces

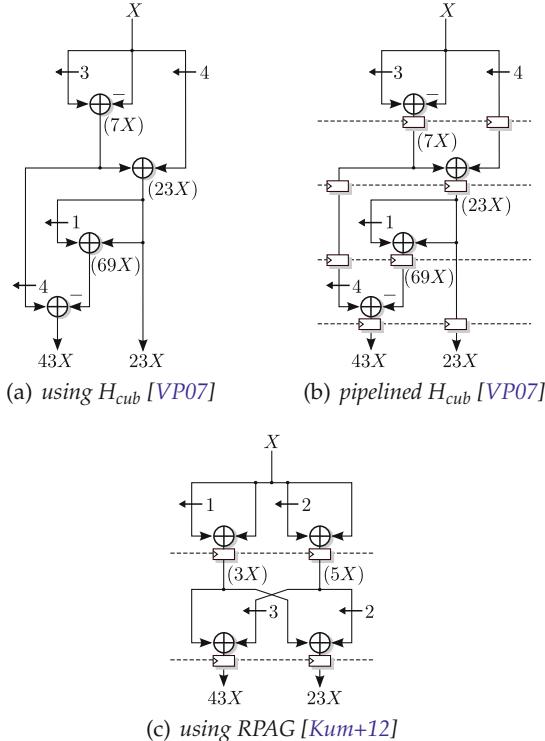


Fig. 12.20 Example PMCM operations for coefficients from the set $\{23, 43\}$.

pipelined adder graphs (PAGs) with less complexity compared to solutions obtained by one of the best MCM heuristics H_{cub} [VP07] and applying an optimal pipeline schedule method [KZ11] afterwards. Optimal ILP-based PMCM methods were proposed in [Kum+13a].

An example is given in Fig. 12.20b, which is a PAG obtained by pipelining the MCM circuit of Fig. 12.20a using an as-soon-as-possible (ASAP) scheduling. Nine additional registers are needed from which five registers are used to balance the pipeline. The PAG solution obtained by RPAG is shown in Fig. 12.20c. It uses the same number of adders but only four registers. Note that on FPGAs, the registers from Fig. 12.20c come for free as they otherwise remain unused. This is in contrast to the five pipeline balance registers of Fig. 12.20b.

On ASICs, pipelined adder graphs are also beneficial for increased performance, but there, carry-save adders (CSAs) may be an alternative for increasing the speed. However, it was shown that there is no 1:1 mapping of CPA-based adder graphs to CSA adder graphs (some CPAs can be replaced

by wires while others require two CSAs). Hence, specialized optimization algorithms for CSA-based adder graphs are required [GW11].

12.5.2 Table-Based Multiple Constant Multiplication

The table-based constant multiplication as discussed in Sect. 12.2 can also be extended to the MCM case. Here, LUTs with identical contents can be shared between different constant multipliers to reduce resources.

At first view, it seems to be unlikely that LUTs with identical contents appear: for an α -input LUT, there are 2^{2^α} possible ways to fill it. However, Faust et al. [FC11] remarked that the maximal number of required LUTs is far less than combinatorially possible. For example, still using 4-input LUTs, only 52 different 1-bit tables (out of $2^{2^4} = 65536$ possible) are sufficient to compute all signed products by all the integer constants from 1 to 2^{12} . The reason for that is again the mathematical structure of such tables. As Fig. 12.21 shows, there is a pattern in each column (quasi-periodic, with strings of zeroes followed by string of ones). This is what explains the reduction of combinatorics compared to a random filling of the tables.

Aksøy et al. tried to maximize this sharing by allowing each constant to vary within a small error interval [AFM15] and then using a 0-1 ILP formulation of the problem. In their experiments, this leads to a reduction of up to 20% in the number of architectural LUTs.

A general result of these works is that on FPGAs, a table-based method provides lower area and better delay for low input word sizes. Kumm et al. [Kum+13a] integrated table-based multipliers in an ILP formulation of the shift-and-add MCM problem.

12.5.3 Sum of Constant Products, Integer Case

The SOPC operation computes

$$R = \sum_{i=0}^{N-1} C_i X_i. \quad (12.44)$$

While MCM is a single input, multiple output circuit, the SOP is a multiple input, single output counterpart. When implemented as shift-and-add, MCM and SOPC operations are algorithmically related, a shift-and-add-based MCM circuit, for a given set of constants, can be converted to a SOPC operation with the same constants by transposing the corresponding signal flow graph (SFG), and vice versa [GD04].

0 × C	00000000000000000000	00000000000000000000
1 × C	0000011000000111001	00000111001010001000111
2 × C	00001100000011110010	00001110010100010001110
3 × C	0001001000010101011	00010101011110011010101
4 × C	0001100000011100100	00011100101000100011100
5 × C	00011110000100011101	001000111100101011000011
6 × C	0010010000101010110	00101010111100110101010
7 × C	00101010001100011111	00110010000110111110001
8 × C	0011000000111001000	00111001010001000111000
9 × C	0011010000001110010001	01000000011101100111111
10 × C	0011100000011101010	01000111100101011000110
11 × C	0100000100101110011	01001110101111100001101
12 × C	0100100001010101100	01010101111001101010100
13 × C	01001110001011100101	01011101000011110011011
14 × C	01010100001100011110	01100100001101111000010
15 × C	0101010001101010111	01101011011000000101001
16 × C	01100000001110010000	01110010100010001110000
17 × C	01100011001111001001	01111001101100010110111
18 × C	0110110010000000010	10000000110110011111110
19 × C	01110010100000111011	1000100000000101000101
20 × C	0111100010001110100	10001111001010110001100
21 × C	0111110010010101101	10010110010100111010011
22 × C	10000010010111000110	10011101011111000011010
23 × C	1000101010100011111	101001001010010001100001
24 × C	10010000010101011000	10101011110011010101000
25 × C	10010101010110010001	10110010111101011101111
26 × C	100101010110010001	10111010111101011101111
27 × C	1001110101100101010	10111010000111100110110
28 × C	1010001011000000011	11000001010001101111101
29 × C	1010100011100111100	11001000011011111000100
30 × C	1010111011001110101	11001111100110000001011
31 × C	1011010011010101110	11010110110000001010010

C = 12345

C = 234567

Fig. 12.21 KCM table internal structure ($\alpha = 5$) for two constants. Each column is stored in one FPGA LUT (see Fig. 12.9b). The three blue columns are identical. The green column is all zero. All other columns (e.g., red and magenta) show a quasi-periodic pattern of strings of zeros followed by strings of ones.

Take, for example, the optimal MCM adder circuit for multiplying with 19 and 43 of Fig. 12.19c which is repeated in Fig. 12.22a. Its SFG representation is shown in Fig. 12.22b. In the SFG, each node with outgoing edges represent branches (filled in white) while nodes with ingoing edges represent adders (filled in black). The edge weights represent multiplicative weights. A minimum adder SOPC circuit can be constructed by transposing a minimum adder SFG of an MCM adder graph. To transpose the SFG, each edge is reversed, which means that inputs become outputs (and vice versa) and branches become adders (and vice versa). This step is illustrated in Fig. 12.22c where the position of the nodes was kept as before resulting in a signal flow from bottom to top. After reorganizing the signal flow from top to bottom, Fig. 12.22d is obtained. The new inputs are renamed as X and Y, resulting that the new input X, which was output 19X before, now contributes with a factor of 19 to the SOPC result while the new input Y (output 43X before) contributes with a factor of 43. The corresponding adder circuit for the corresponding SOPC circuit is shown in Fig. 12.22e.

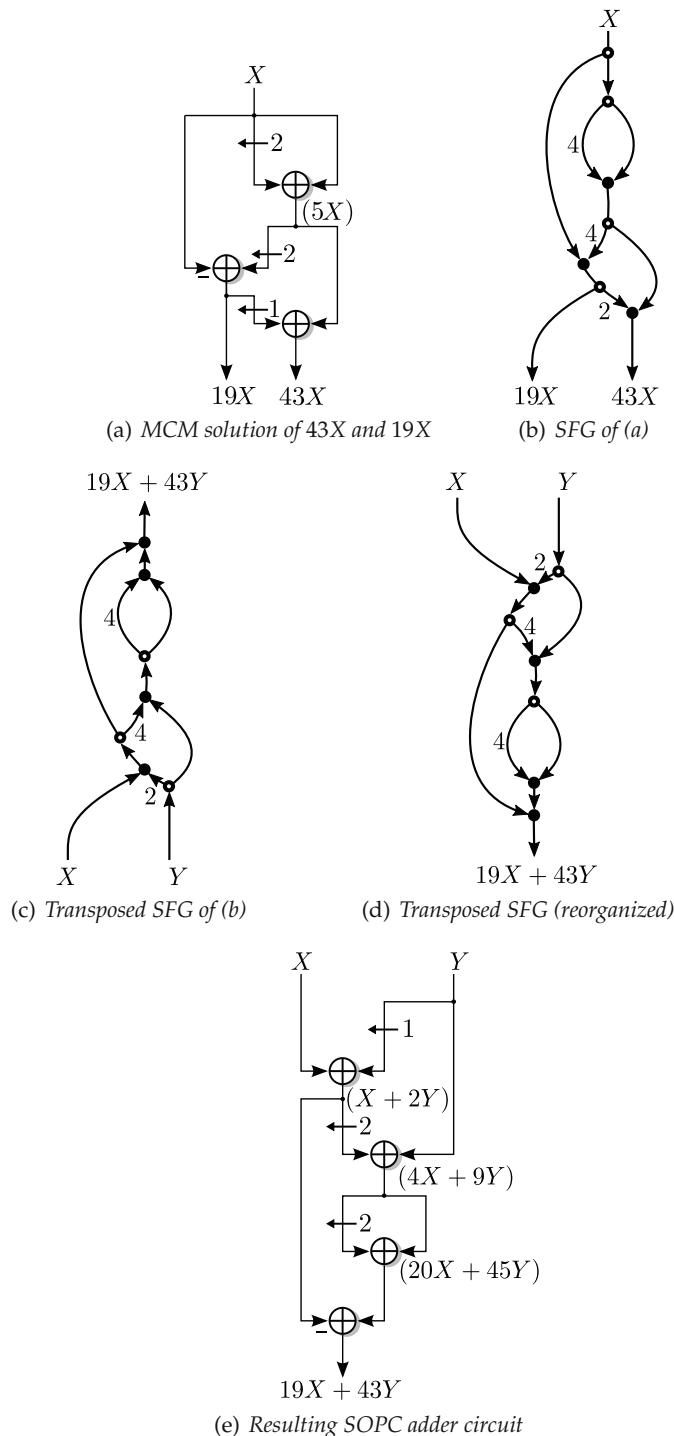


Fig. 12.22 Example of obtaining an SOPC adder graph from an MCM adder graph.

12.5.4 Constant Matrix-Vector Multiplication

The MCM as well as the SOPC operations can be further extended to the constant matrix multiplication (CMM) operation by allowing several inputs and outputs. Take, for example, the complex multiplication

$$(X_r + jX_i) \cdot (19 + j43) = \underbrace{19X_r - 43X_i}_{=Y_r} + j(\underbrace{43X_r + 19X_i}_{=Y_i}) \quad (12.45)$$

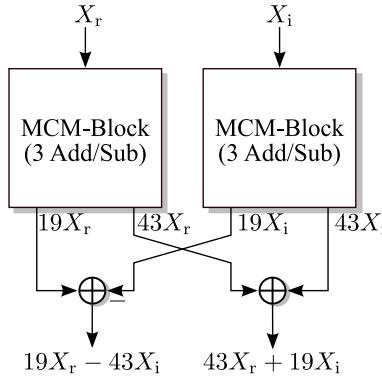
which can be represented in matrix form as

$$\begin{pmatrix} Y_r \\ Y_i \end{pmatrix} = \begin{pmatrix} 19 & -43 \\ 43 & 19 \end{pmatrix} \cdot \begin{pmatrix} X_r \\ X_i \end{pmatrix} = \begin{pmatrix} 19X_r - 43X_i \\ 43X_r + 19X_i \end{pmatrix}. \quad (12.46)$$

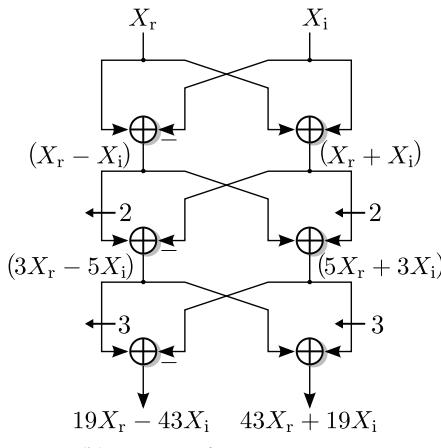
We could use the MCM operation of Fig. 12.19c for each of the inputs and add another two adders to compute the result as shown in Fig. 12.23a. This solution would require eight additions in total. Alternatively, we could use two SOPC circuits, similar to Fig. 12.22e, leading to a similar adder count. However, as demonstrated in Fig. 12.23b, by sharing intermediate results a solution with only six adders is possible.

looseness-1 The advantage of directly optimizing constant matrix multiplication (CMM) circuits instead of using an MCM optimization multiple times was first demonstrated by Hartley [Har91; Har96] and later formalized by Dempster et al. [DGC03]. They extended their Bull and Horrocks Modified (BHM) MCM optimization algorithm [DM95] to solve the CMM problem. This inspired many other authors to develop CMM optimization methods. Another CMM heuristic based on an MCM heuristic [GW02] was proposed by Gustafsson et al. [GOW04]. They transferred the CMM problem to a graph representation in which a minimum spanning tree (MST) has to be found. This MST yields a new matrix with reduced complexity which is used for the next iteration until the matrix only contains ones and zeros. A CMM method based on CSE was introduced by Macleod and Dempster [MD04]. They represent the CMM instance by using an $i \times j \times k$ matrix which contains the SD value at bit position k for the constant of row i in column j . Then, two-term subexpressions that occur most often are searched and subsequently eliminated. Another method for optimizing CMM using a genetic algorithm (GA) was proposed by Kinane et al. [KMO06b; KMO06a]. Permutations from different SD representations are taken to extract valid SOPC sub-terms. These are then combined and selected by a GA. The RPAG algorithm for the PMCM problem was recently extended to solve the CMM problem [KHZ17].

In general, a CMM that multiplies by a constant $M \times N$ matrix \mathbf{C} has M outputs and N inputs. Transposing the SFG of a CMM that multiplies



(a) CMM with two MCM operations



(b) Optimized CMM operation

Fig. 12.23 Example CMM operation with constant $19 + j43$.

by a constant $M \times N$ matrix \mathbf{C} results in a CMM that multiplies with the transposed matrix \mathbf{C}^T , which has size $N \times M$. It was shown in [Gus07b] that transposing an adder graph with N_A adders results in a graph with $N_A - N + M$ adders. This implies that transposing an adder graph with $N = M$ (including $N = M = 1$) does not change the adder cost. Another interesting property was observed in [SGG20]: Graph-based CMM algorithms typically scale better with rows (M) than with columns (N). Hence, in case there are more columns than rows ($N > M$), it is usually faster to solve CMM on the transposed matrix and to transpose the resulting SFG [SGG20].

12.5.5 Table-Based Sum of Products of Fixed-Point Inputs by Real Constants

When the constant multipliers are implemented using the KCM technique, there is a new optimization opportunity (in addition to the LUT sharing opportunity already studied in Sect. 12.5.2): the fusion of the double summation in a single bit heap. Actually, considering the SOPC as a single operation has two advantage: a performance advantage due to a single global optimization of the bit heap and an accuracy advantage due to a single rounding process with fewer rounding error (which may itself entail a reduced number of guard bits, hence improved performance).

Let us study this in the more generally useful case when the constants are provided as real numbers:

$$R \approx \sum_{i=0}^{N-1} C_i X_i \quad \text{where } \forall i \ X_i \text{ is a sfix}(0, \ell_i) \text{ number} \quad (12.47)$$

In the KCM algorithm, each X_i is decomposed into α -bit chunks X_{ik} as per (12.26), and the SOPC result can be obtained by computing a double sum:

$$R \approx \sum_{i=0}^{N-1} \sum_{k=0}^{D_i-1} 2^{k\alpha+\ell_i} C_i X_{ik} \quad (12.48)$$

where each term $2^{k\alpha+\ell_i} C_i X_{ik}$ will be precomputed and stored in a table $T_{ik}(X_{ik})$:

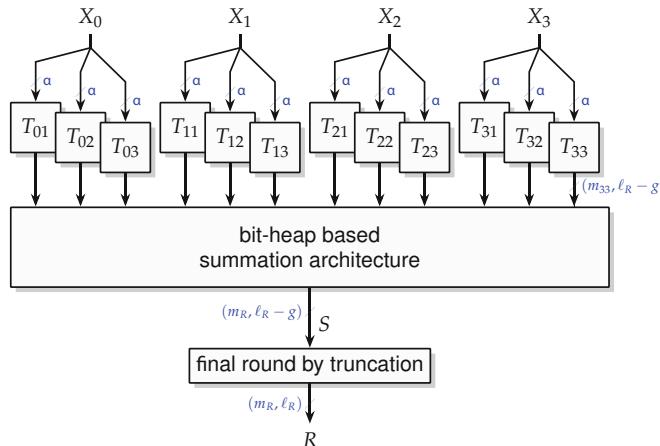


Fig. 12.24 KCM-based SOPC architecture for $N = 4$, each input being split into 3 chunks, with one single rounding.

$$R = \left[\sum_{i=0}^{N-1} \sum_{k=0}^{D_i-1} 2^{k\alpha+\ell_i} T_{ik} \right]_{\ell_R}. \quad (12.49)$$

As Fig. 12.24 illustrates, this double sum can be computed as a single bit heap. Besides, there is no need to round each individual KCM as per Sect. 12.3: All the individual table rounding errors can be merged in a single, global error analysis.

Here, all the tables should share the same output LSB, for a reason already invoked several times: If one of the tables is more accurate than another one, then it is accurate in vain. It is natural to express this common LSB as $\ell_R - g$ (see Fig. 12.24), where g is a number of guard bits. Hence, all the tables should share the same value of g . In other terms, table T_{ik} holds

$$T_{ik}(X_{ik}) = \left[2^{k\alpha+\ell_i} C X_{ik} \right]_{\ell_R-g}. \quad (12.50)$$

Figure 12.25 shows the bit heap of a KCM-based SOPC.

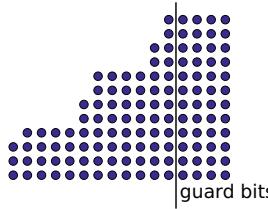


Fig. 12.25 An example bit heap for the FixSOPC represented in Fig. 12.24. Specifically, this is the bit heap of a 4th-order half-sine FIR filter with $m_X = 0$, $\ell_X = \ell_R = -12$. Each line corresponds to the output of a T_{ik} table. Here each X_i is split into 3 chunks of 5 bits each (in the SOPC of a FIR filter, all the X_i have the same format), and $g = 4$ ensures last-bit accuracy with respect to the infinitely accurate filter.

Now, the rounding errors of each T_{ik} add up into an overall SOPC error, out of which the value of the shared g can be computed. We could just generalize the worst-case error analysis of Sect. 12.3, which would quite straightforwardly lead to replacing (12.36) with

$$g > \log_2(ND). \quad (12.51)$$

For instance, for the half-sine filter whose bit heap is represented on Fig. 12.25, we have $N = 4$ and each product is implemented through three tables; hence, $g = 4$.

The FloPoCo code for `FixRealSOPC`, following Sect. 12.3.3.2, first invokes, for each constant C_i , a method that returns the maximum error $\overline{\delta}_{\text{round},i}^{(\text{gulp})}$ that will be entailed by a multiplier by this constant. The generator sums

these errors into an overall rounding error bound $\overline{\delta_{\text{round}}^{(\text{gulp})}}$ and then uses this sum to compute the value of g that will enable last-bit accuracy:

$$g > 1 + \log_2 \overline{\delta_{\text{round}}^{(\text{gulp})}}. \quad (12.52)$$

Once the proper value of g has been determined, the generator may proceed with the creation of the bit heap, the actual construction of the multipliers (throwing their table outputs to this bit heap), the generation of the compressor tree, and the final rounding by truncation (thanks to an half-ulp 2^{ℓ_R-1} added to one of the tables).

Filters and SOPC

Digital filters typically use SOPC operators, but their design is more complex for two main reasons:

- In recursive filters, the SOPC output is reinjected into the summation, and this (infinite) recursion requires a more complex error analysis than the one sketched above.
- The specification of a filter is often a frequency response, which gives some freedom in the choice of the coefficients.

These issues (and others) are addressed in Chap. 23.

12.6 Other FPGA-Specific Techniques

12.6.1 Constant Multiplication Using DSP Blocks on FPGAs

Since FPGAs provide embedded multipliers and even DSP blocks (see Sect. 4.2), they can also be used for constant multiplications. In case their amount is limited, i. e., there are fewer DSP blocks available than needed for the constant multiplications, a hybrid approach can be used which uses both the DSP blocks as well as the shift-and-add algorithms described above. One possibility for small constants and data word sizes was proposed in [Mer+18]. It is based on previous work proposed in [Xil17], where each DSP block computes two constant multiplications by zero padding the arguments and splitting the results accordingly. With that, typically two operations of the form $X \times Z$ and $Y \times Z$ can be performed with a single DSP block. When Z is a constant, the idea in [Mer+18] is to map one nonzero digit of the constant to the post-adder of the DSP which increases the coefficient word size. For larger constants, the idea proposed in [KZ12] is to replace the most complex coefficients by DSP blocks. The DSP results may still be shared in MCM or large constant multiplications. With *large* we mean that

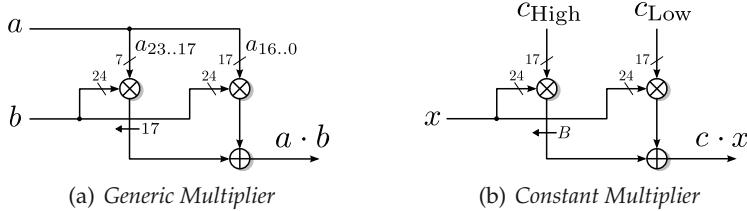


Fig. 12.26 A single-precision mantissa multiplier using DSP48E blocks on a Xilinx Virtex 6 FPGA.

the coefficient requires a larger word size than the DSP block input word size.

Take, for example, the 17×24 bit multipliers (unsigned) of the DSP48(E) blocks of Xilinx' Virtex 5/6/7 FPGAs. A 24×24 bit multiplication requires to cascade two DSP blocks as already discussed in Chap. 8 and shown in Fig. 12.26a. One argument is split into two smaller words fitting the input word size of 17 bit and the results are bit-shifted and added by using the post-adder of the DSP block. In the case when this argument is a constant, this constant can be represented as

$$c = c_{\text{Low}} \pm 2^B c_{\text{High}} \quad (12.53)$$

leading to the structure shown in Fig. 12.26b.

In principle, there are many combinations of c_{Low} and c_{High} as long as the constant is less than the total input word size (which is $2 \cdot 17 \text{ bit} = 34 \text{ bit}$ in this example). To illustrate this, consider the 24-bit constant 11508221 which can be split into the following $c_{\text{Low}}/c_{\text{High}}$ combinations:

$$11508221 = 104957 + 2^{17} \cdot 87 \quad (12.54)$$

$$= 39421 + 2^{16} \cdot 175 \quad (12.55)$$

$$= 6653 + 2^{15} \cdot 351 \quad (12.56)$$

$$= 39421 + 2^{15} \cdot 350 \quad (12.57)$$

$$= 72189 + 2^{15} \cdot 349 \quad (12.58)$$

This degree of freedom can be used to find $c_{\text{Low}}/c_{\text{High}}$ combinations which can be shared among different output coefficients, or where one of the constants has a low cost when implemented as shift-and-add.

A systematic exploration was integrated in the (P)MCM search of the RPAG algorithm [KZ12]. Figure 12.27 shows example solutions for a five-coefficient FIR filter in single precision using the conventional method (Fig. 12.27a) and an optimized RPAG solution (Fig. 12.27b). One can observe that

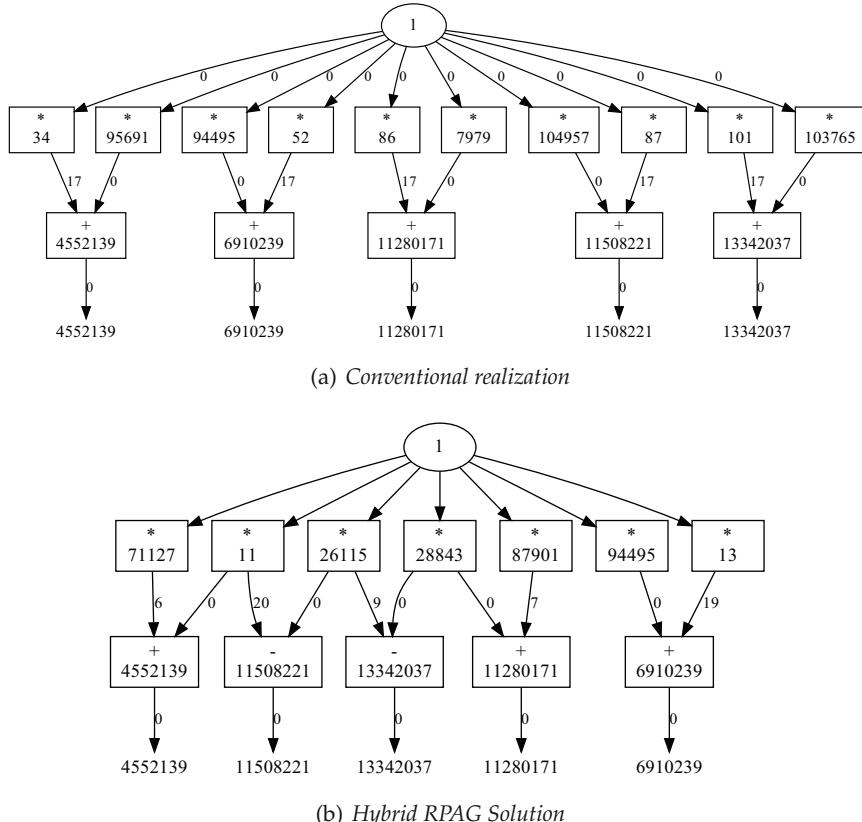


Fig. 12.27 Hybrid adder/multiplier graph for a 5-coefficient benchmark filter.

many of the DSP results can be shared to reduce the DSP count from ten to seven.

The DSP blocks of recent Intel FPGAs have inputs of identical size, which makes the previous technique less relevant. They also include a small coefficient memory that can hold up to 8 constant coefficients [Int18b]. Such a DSP block can be used as a reconfigurable constant multiplier, a use case that we consider now.

12.6.2 Reconfigurable Constant Multiplication

There are situations where a multiplication has to be performed not with a constant but with a constant from a small set of values (i.e., $C \in \{C_0, C_1, \dots, C_N\}$). The coefficient is selected from a control input having

$\lceil \log_2(N) \rceil$ bits. This type of circuit is called reconfigurable constant coefficient multiplier (RCCM) or *time-multiplexed* constant multiplier. It appears, e.g., in sequential implementations of digital filters for the different coefficients (see Chap. 23) or neural networks, where weights have to be changed from time to time (see Chap. 24). One popular solution for this is to use constant multipliers based on shift-and-add and to introduce multiplexer (MUX)es to the corresponding adder graph [TCW01; THP07; Möl+17; Möl17].

Figure 12.28 shows a simple example of an RCCM that was designed for multiplication by $C \in \{45, 19\}$. When all MUXes are set to the left input, the result will be computed as $(2^2 + 1)(2^3 + 1)X = 45X$ while it computes $(2^1 + 1)X + 2^4 X = 19X$ for the other case. The control input is directly connected to the select signal of the MUX (not shown in Fig. 12.28).

One way to design such circuit is by fusing several adder graphs, each representing a constant, into a single reconfigurable adder graph. This problem has been addressed for application-specific integrated circuits (ASICs) [THP07; Möl+18] as well as for FPGAs [Möл+17; Möл17]. The algorithms can in principle handle adder graphs for MCM or CMM. Here, the index selects a vector (MCM) or a matrix (CMM) of constants. We do not go further into the details here and refer the reader to the literature [Möл17].

On FPGAs, an alternative is using reconfigurable LUTs to reconfigure a KCM-based constant multiplier (see Sect. 12.2). This will take a couple of clock cycles (32 on recent Xilinx FPGAs). Hence, this is attractive for applications in which the coefficients only have to be changed from time to time, like the network weights in convolutional neural networks [Har19].

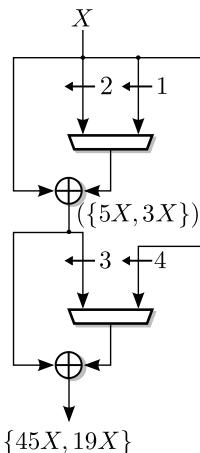


Fig. 12.28 Example of an RCCM able to multiply with $C \in \{45, 19\}$.

12.7 Conclusion: Choosing the Best Constant Multiplication Technique in a Given Context

To sum up, there is plenty of choice in terms of constant multiplication.

When working with ASICs, the best solution is to use a shift-and-add method. Recent works can compute optimal multipliers for constants of most practical sizes, and very good heuristics exist when the optimal algorithms fail.

When working with FPGAs, as a rule of thumb, KCM is better, except for really simple constants where shift-and-add can be more efficient. In some cases, the choice is obvious: For instance, to evaluate a floating-point exponential in Chap. 22, we have to multiply an exponent (a small integer) by $\log(2)$, and we need many more bits on the result—this is a case for KCM, as we would need to consider many bits of the constant.

In most cases, however, the final choice should probably be done on a trial and error basis. This choice should actually be performed by the generator, based on an evaluation of the performance and resource consumption of the operator being generated. At the time of writing this book, FloPoCo does not yet offer this functionality.

References

- [802.15.4] IEEE Standard for Information technology— Telecommunications and information exchange between systems— Local and metropolitan area networks— Specific requirements— Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs). 2006. (cit. on p. [386](#)).
- [AFM15] Levent Aksoy, Paulo Flores, and José Monteiro. “Approximation of Multiple Constant Multiplications Using Minimum Look-Up Tables on FPGA”. In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2015, pp. 2884–2887. (cit. on p. [407](#)).
- [AGF08] Levent Aksoy, E. Gunes, and Paulo Flores. “An Exact Breadth-First Search Algorithm for the Multiple Constant Multiplications Problem”. In: *NORCHIP* (2008), pp. 41–46. (cit. on pp. [373, 405](#)).
- [AGF10] Levent Aksoy, E.O. Günes, and Paulo Flores. “Search Algorithms for the Multiple Constant Multiplications Problem: Exact and Approximate”. In: *Microprocessors and Microsystems* 34.5 (2010), pp. 151–162. (cit. on pp. [373, 405](#)).

- [AHS76] Ehud Artzy, James A. Hinds, and Harry J. Saal. "A Fast Division Technique for Constant Divisors". In: *Communications of the ACM* 19 (1976), pp. 98–101. (cit. on p. 395).
- [AJU76] A. V. Aho, S. C. Johnson, and J. D. Ullman. "Code generation for expressions with common subexpressions (Extended Abstract)". In: *ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages (POPL)*. 1976. (cit. on p. 369).
- [Aks+07] Levent Aksoy, Eduardo da Costa, Paulo Flores, and José Monteiro. "Optimization of Area in Digital FIR Filters using Gate-Level Metrics". In: *Design Automation Conference (DAC)*. ACM/IEEE, 2007, pp. 420–423. (cit. on p. 381).
- [Aks+08] Levent Aksoy, Eduardo da Costa, Paulo Flores, and José Monteiro. "Exact and Approximate Algorithms for the Optimization of Area and Delay in Multiple Constant Multiplications". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.6 (2008), pp. 1013–1026. (cit. on p. 404).
- [Aks+11] Levent Aksoy, Eduardo Costa, Paulo Flores, and José Monteiro. "Design of Low-Power Multiple Constant Multiplications Using Low-Complexity Minimum Depth Operations". In: *Great Lakes Symposium on VLSI (GLSVLSI)*. ACM, 2011, pp. 79–84. (cit. on p. 373).
- [Aks09] Levent Aksoy. "Optimization Algorithms for the Multiple Constant Multiplications Problem". PhD thesis. Istanbul Technical University, 2009. (cit. on p. 373).
- [Avi61] Algirdas Antanas Avižienis. "Signed-Digit Number Representations for Fast Parallel Arithmetic". In: *IRE Transactions on Electronic Computers* EC-10.3 (1961), pp. 389–400. (cit. on p. 368).
- [BDM08] Nicolas Brisebarre, Florent de Dinechin, and Jean-Michel Muller. "Integer and Floating-Point Constant Multipliers for FPGAs". In: *International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE, 2008, pp. 239–244. (cit. on p. 381).
- [Boo51] Andrew D. Booth. "A Signed Binary Multiplication Technique". In: *The Quarterly Journal of Mechanics and Applied Mathematics* (1951), pp. 236–240. (cit. on p. 368).
- [Cha94] Ken D. Chapman. "Fast Integer Multipliers Fit in FPGAs". In: *Electronic Design News* (1994). (cit. on p. 384).
- [CS84] Peter Cappello and Kenneth Steiglitz. "Some Complexity Issues in Digital Signal Processing". In: *IEEE Transactions on Acoustics, Speech and Signal Processing* 32.5 (1984), pp. 1037–1041. (cit. on p. 369).
- [DDK00] Süleyman S. Demirsoy, Andrew G. Dempster, and Izzet Kale. "Transition Analysis on FPGA for Multiplier-Block Based FIR Filter Structures". In: *International Symposium on Circuits and Systems (ISCAS)*. Vol. 2. IEEE, 2000, pp. 862–865. (cit. on p. 373).

- [DDK02] Andrew G. Dempster, Süleyman S. Demirsoy, and Izzet Kale. "Designing Multiplier Blocks with Low Logic Depth". In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2002, pp. 773–776. (cit. on p. [373](#)).
- [DGC03] Andrew G. Dempster, Oscar Gustafsson, and Jeffrey O. Coleman. "Towards an Algorithm for Matrix Multiplier Blocks". In: *European Conference on Circuit Theory and Design (ECCTD)*. 2003, pp. 1–4. (cit. on p. [410](#)).
- [Din+19] Florent de Dinechin, Silviu-Ioan Filip, Luc Forget, and Martin Kumm. "Table-Based versus Shift-And-Add Constant Multipliers for FPGAs". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2019. (cit. on pp. [387](#), [389](#)).
- [DIZ07] Vassil Dimitrov, Laurent Imbert, and Andrew Zakaluzny. "Multiplication by a Constant is Sublinear". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2007, pp. 261–268. (cit. on p. [400](#)).
- [DM94] Andrew G. Dempster and Malcolm D. Macleod. "Constant Integer Multiplication Using Minimum Adders". In: *IEE Proceedings of Circuits, Devices and Systems* 141.5 (1994), pp. 407–413. (cit. on pp. [370](#), [399](#)).
- [DM95] Andrew G. Dempster and Malcolm D. Macleod. "Use of Minimum-Adder Multiplier Blocks in FIR Digital Filters". In: *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 42.9 (1995), pp. 569–577. (cit. on pp. [402](#), [410](#)).
- [FC10] Mathias Faust and Chip-Hong Chang. "Minimal Logic Depth Adder Tree Optimization for Multiple Constant Multiplication". In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2010, pp. 457–460. (cit. on pp. [369](#), [373](#)).
- [FC11] Mathias Faust and Chip-Hong Chang. "Bit-parallel Multiple Constant Multiplication using Look-Up Tables on FPGA". In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2011, pp. 657–660. (cit. on p. [407](#)).
- [GD04] Oscar Gustafsson and Andrew G. Dempster. "On the Use of Multiple Constant Multiplication in Polyphase FIR Filters and Filter Banks". In: *Nordic Signal Processing Symposium (NORSIG)*. 2004, pp. 53–56. (cit. on p. [407](#)).
- [GDW02] Oscar Gustafsson, Andrew G. Dempster, and Lars Wanhammar. "Extended Results for Minimum-Adder Constant Integer Multipliers". In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2002, pp. 73–76. (cit. on p. [370](#)).
- [GJD09] Oscar Gustafsson, Kenny Johansson, and Linda S. DeBrunner. "Techniques for Avoiding Sign-Extension in Multiple Constant Multiplication". In: *Asilomar Conference on Signals, Circuits and Systems*. IEEE, 2009, pp. 740–743. (cit. on p. [381](#)).

- [Gla78] J. W. L. Glaisher. "Periods of Reciprocals of Integers Prime to 10". In: *Proceedings of the Cambridge Philosophical Society* 3 (1878), pp. 185–206. (cit. on p. 395).
- [GOW04] Oscar Gustafsson, Henrik Ohlsson, and Lars Wanhammar. "Low-Complexity Constant Coefficient Matrix Multiplication Using a Minimum Spanning Tree Approach". In: *6th Nordic Signal Processing Symposium (NORSIG)*. 2004, pp. 141–144. (cit. on p. 410).
- [GQ10] Oscar Gustafsson and Fahad Qureshi. "Addition Aware Quantization for Low Complexity and High Precision Constant Multiplication". In: *IEEE Signal Processing Letters* 17.2 (2010), pp. 173–176. (cit. on pp. 389, 395, 398).
- [Gurobi] Gurobi Optimization Inc. *Gurobi Website*. 2022. URL: <http://www.gurobi.com>. (cit. on p. 378).
- [Gus+06] Oscar Gustafsson, Andrew G. Dempster, Kenny Johansson, and Malcolm D. Macleod. "Simplified Design of Constant Coefficient Multipliers". In: *Circuits, Systems, and Signal Processing* 25.2 (2006), pp. 225–251. (cit. on pp. 370, 379, 399).
- [Gus07a] Oscar Gustafsson. "A Difference Based Adder Graph Heuristic for Multiple Constant Multiplication Problems". In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2007, pp. 1097–1100. (cit. on pp. 402, 404).
- [Gus07b] Oscar Gustafsson. "Lower Bounds for Constant Multiplication Problems". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 54.11 (2007), pp. 974–978. (cit. on pp. 373, 399, 400, 405, 411).
- [Gus08] Oscar Gustafsson. "Towards Optimal Multiple Constant Multiplication: A Hypergraph Approach". In: *Asilomar Conference on Signals, Circuits and Systems*. IEEE, 2008, pp. 1805–1809. (cit. on pp. 373, 404).
- [GV23] Remi Garcia and Anastasia Volkova. "Toward the Multiple Constant Multiplication at Minimal Hardware Cost". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* (2023), pp. 1–13. (cit. on pp. 381, 387, 389, 405).
- [GVK22] Rémi Garcia, Anastasia Volkova, and Martin Kumm. "Truncated Multiple Constant Multiplication with Minimal Number of Full Adders". In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2022, pp. 263–267. (cit. on pp. 381, 387, 389, 405).
- [GW02] Oscar Gustafsson and Lars Wanhammar. "A Novel Approach to Multiple Constant Multiplication Using Minimum Spanning Trees". In: *IEEE Midwest Symposium on Circuits and Systems (MWSCAS)*. Vol. 3. 2002. (cit. on p. 410).
- [GW11] Oscar Gustafsson and Lars Wanhammar. "Low-Complexity and High-Speed Constant Multiplications for Digital Filters

- Using Carry-Save Arithmetic". In: *Digital Filters*. InTech, 2011. (cit. on p. 407).
- [Har+19] Martin Hardieck, M Kumm, Konrad Möller, and Peter Zipf. "Reconfigurable Convolutional Kernels for Neural Networks on FPGAs". In: *International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM, 2019, pp. 43–52. (cit. on p. 417).
- [Har91] Richard Hartley. "Optimization of Canonic Signed Digit Multipliers for Filter Design". In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1991, pp. 1992–1995. (cit. on p. 410).
- [Har96] Richard I. Hartley. "Subexpression Sharing in Filters Using Canonic Signed Digit Multipliers". In: *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 43.10 (1996), pp. 677–688. (cit. on pp. 369, 410).
- [Hwa79] Kai Hwang. *Computer Arithmetic: Principles, Architecture, and Design*. Wiley-Interscience, 1979. (cit. on p. 368).
- [Int18b] *Intel Stratix 10 Variable Precision DSP Blocks User Guide*. Intel Corporation. 2018. (cit. on p. 416).
- [JBD17] Miriam Guadalupe Cruz Jimenez, Uwe Meyer Baese, and Gordana Jovanovic Dolecek. "Theoretical Lower Bounds for Parallel Pipelined Shift-and-Add Constant Multiplications with N-Input Arithmetic Operators". In: *EURASIP Journal on Advances in Signal Processing* 2017.1 (2017), pp. 1–13. (cit. on p. 373).
- [JGW05] Kenny Johansson, Oscar Gustafsson, and Lars Wanhammar. "A Detailed Complexity Model for Multiple Constant Multiplication and an Algorithm to Minimize the Complexity". In: *European Conference on Circuit Theory and Design*. Vol. 3. 2005, III/465–III/468 vol. 3. (cit. on p. 381).
- [JGW07] Kenny Johansson, Oscar Gustafsson, and Lars Wanhammar. "Bit-Level Optimization of Shift-and-Add Based FIR Filters". In: *International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, 2007, pp. 713–716. (cit. on p. 381).
- [Joh08] Kenny Johansson. "Low Power and Low Complexity Shift-and-Add Based Computations". PhD thesis. Linköping Studies in Science and Technology, 2008. (cit. on p. 373).
- [KHZ17] Martin Kumm, Martin Hardieck, and Peter Zipf. "Optimization of Constant Matrix Multiplication with Low Power and High Throughput". In: *IEEE Transactions on Computers* 66.12 (2017), pp. 2072–2080. (cit. on p. 410).
- [KMO06a] Andrew Kinane, Valentin Muresan, and Noel O'Connor. "Optimisation of Constant Matrix Multiplication Operation Hardware Using a Genetic Algorithm". In: *Lecture Notes in Computer Science*. Springer, 2006, pp. 296–307. (cit. on p. 410).
- [KMO06b] Andrew Kinane, Valentin Muresan, and Noel O'Connor. "Towards an Optimised VLSI Design Algorithm for the Constant Matrix Multiplication Problem". In: *International Symposium on*

- Circuits and Systems (ISCAS)*. IEEE, 2006, pp. 5111–5114. (cit. on p. 410).
- [Knu97] Donald Knuth. *The Art of Computer Programming, vol.2: Seminumerical Algorithms*. 3rd ed. Addison Wesley, 1997. (cit. on p. 368).
- [Kum+12] Martin Kumm, Peter Zipf, Mathias Faust, and Chip-Hong Chang. “Pipelined Adder Graph Optimization for High Speed Multiple Constant Multiplication”. In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2012, pp. 49–52. (cit. on pp. 373, 405).
- [Kum+13a] Martin Kumm, Diana Fanghänel, K Möller, Peter Zipf, and Uwe Meyer-Baese. “FIR Filter Optimization for Video Processing on FPGAs”. In: *Springer EURASIP Journal on Advances in Signal Processing* (2013), pp. 1–18. (cit. on pp. 373, 404, 406, 407).
- [Kum+13b] Martin Kumm, Martin Hardieck, Jens Willkomm, Peter Zipf, and Uwe Meyer-Baese. “Multiple Constant Multiplication with Ternary Adders”. In: *International Conference on Field Programmable Logic and Application (FPL)*. 2013, pp. 1–8. (cit. on pp. 373, 380).
- [Kum+16] Martin Kumm, Oscar Gustafsson, Mario Garrido, and Peter Zipf. “Optimal Single Constant Multiplication using Ternary Adders”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 65.7 (2016), pp. 928–932. (cit. on pp. 373, 378).
- [Kum15] Martin Kumm. “Multiple Constant Multiplication Optimizations for Field Programmable Gate Arrays”. PhD thesis. Springer Wiesbaden, 2015. (cit. on pp. 376, 404, 405).
- [Kum18] Martin Kumm. “Optimal Constant Multiplication using Integer Linear Programming”. In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2018. (cit. on pp. 373, 405).
- [KZ11] Martin Kumm and Peter Zipf. “High Speed Low Complexity FPGA-Based FIR Filters Using Pipelined Adder Graphs”. In: *International Conference on Field-Programmable Technology (FPT)*. IEEE, 2011, pp. 1–4. (cit. on pp. 405, 406).
- [KZ12] Martin Kumm and Peter Zipf. “Hybrid Multiple Constant Multiplication for FPGAs”. In: *International Conference on Electronics, Circuits and Systems, (ICECS)*. 2012, pp. 556–559. (cit. on pp. 414, 415).
- [Li85] Shuo-Yen Robert Li. “Fast Constant Division Routines”. In: *IEEE Transactions on Computers* C-34.9 (1985), pp. 866–869. (cit. on p. 395).
- [LYM14] Xin Lou, Ya Jun Yu, and Pramod Kumar Meher. “High-Speed Multiplier Block Design Based on Bit-Level Critical Path Optimization”. In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2014, pp. 1308–1311. (cit. on p. 381).

- [LYM15] Xin Lou, Ya Jun Yu, and Pramod Kumar Meher. "Fine-Grained Critical Path Analysis and Optimization for Area-Time Efficient Realization of Multiple Constant Multiplications". In: *IEEE Transactions on Circuits and Systems I* 62.3 (2015), pp. 863–872. (cit. on p. [381](#)).
- [MD04] Malcolm D. Macleod and Andrew G. Dempster. "Common Subexpression Elimination Algorithm for Low-Cost Multiplierless Implementation of Matrix Multipliers". In: *Electronics Letters* 40.11 (2004), pp. 651–652. (cit. on p. [410](#)).
- [Mer+18] Ahmed Can Mert, Hasan Azgin, Ercan Kalalı, and İlker Hamzaoglu. "Efficient multiple constant multiplication using DSP blocks in FPGA". In: *International Conference on Field Programmable Logic and Application (FPL)*. IEEE, 2018. (cit. on p. [414](#)).
- [Mey+06] Uwe Meyer-Baese, Jiajia Chen, Chip Hong Chang, and Andrew G. Dempster. "A Comparison of Pipelined RAG-n and DA FPGA-based Multiplierless Filters". In: *IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. 2006, pp. 1555–1558. (cit. on p. [405](#)).
- [Mey14] Uwe Meyer-Baese. *Digital Signal Processing with Field Programmable Gate Arrays*. 4th ed. Springer, 2014. (cit. on p. [368](#)).
- [Möl+17] Konrad Möller, Martin Kumm, Marco Kleinlein, and Peter Zipf. "Reconfigurable Constant Multiplication for FPGAs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36.6 (2017), pp. 927–937. (cit. on pp. [373](#), [417](#)).
- [Möl+18] Konrad Möller, Martin Kumm, Mario Garrido, and Peter Zipf. "Optimal Shift Reassignment in Reconfigurable Constant Multiplication Circuits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.3 (2018), pp. 710–714. (cit. on p. [417](#)).
- [Möl17] Konrad Möller. "Run-time Reconfigurable Constant Multiplication on Field Programmable Gate Arrays". PhD thesis. Kassel University Press, 2017. (cit. on pp. [373](#), [417](#)).
- [PSC96] M. Potkonjak, M. B. Srivastava, and A. P. Chandrakasan. "Multiple Constant Multiplications: Efficient and Versatile Framework and Algorithms for Exploring Common Subexpression Elimination". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 15.2 (1996), pp. 151–165. (cit. on p. [369](#)).
- [SGG20] Narges Mohammadi Sarband, Oscar Gustafsson, and Mario Garrido. "Using Transposition to Efficiently Solve Constant Matrix-Vector Multiplication and Sum of Product Problems". In: *Journal of Signal Processing Systems* 54.11 (July 2020), pp. 1–15. (cit. on p. [411](#)).

- [SP94] P. Srinivasan and F.E. Petry. "Constant-division algorithms". In: *IEE Proc. Computers and Digital Techniques* 141.6 (1994), pp. 334–340. (cit. on p. 395).
- [SY11] Dong Shi and Ya Jun Yu. "Design of Linear Phase FIR Filters With High Probability of Achieving Minimum Number of Adders". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 58.1 (2011), pp. 126–136. (cit. on p. 373).
- [TCW01] Richard H. Turner, Tim Courtney, and Roger Woods. "Implementation of Fixed DSP Functions using the Reduced Coefficient Multiplier". In: *IEEE International Conference on Acoustics, Speech, and Signal Processing* 2 (2001), pp. 881–884. (cit. on p. 417).
- [THP07] Peter Tummeltshammer, James C. Hoe, and Markus Püschel. "Time-Multiplexed Multiple-Constant Multiplication". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.9 (2007), pp. 1551–1563. (cit. on p. 417).
- [TMJ14] David E. Troncoso Romero, Uwe Meyer-Baese, and Gordana Jovanovic Dolecek. "On the Inclusion of Prime Factors to Calculate the Theoretical Lower Bounds in Multiplierless Single Constant Multiplications". In: *EURASIP Journal on Advances in Signal Processing* (2014). (cit. on p. 373).
- [TN10] Jason Thong and Nicola Nicolici. "A Novel Optimal Single Constant Multiplication Algorithm". In: *Design Automation Conference (DAC)*. ACM/IEEE, 2010. (cit. on p. 370).
- [TN11] Jason Thong and Nicola Nicolici. "An Optimal and Practical Approach to Single Constant Multiplication". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.9 (2011), pp. 1373–1386. (cit. on pp. 370, 399).
- [Toc58] Keith D. Tocher. "Techniques of Multiplication and Division for Automatic Binary Computers". In: *The Quarterly Journal of Mechanics and Applied Mathematics* 11.3 (1958), pp. 364–384. (cit. on p. 368).
- [Vol+19] Anastasia Volkova, Matei Istoan, Florent de Dinechin, and Thibault Hilaire. "Towards Hardware IIR Filters Computing Just Right: Direct Form I Case Study". In: *IEEE Transactions on Computers* 68.4 (2019). (cit. on p. 393).
- [VP07] Yevgen Voronenko and Markus Püschel. "Multiplierless Multiple Constant Multiplication". In: *ACM Transactions on Algorithms* 3.2 (2007). (cit. on pp. 370, 402, 404, 406).
- [Wir04] Michael J. Wirthlin. "Constant Coefficient Multiplication Using Look-Up Tables". In: *Journal of VLSI Signal Processing* 36.1 (2004), pp. 7–15. (cit. on pp. 384, 385).
- [Xil17] Yao Fu, Ephrem Wu, Ashish Sirasao, Sedny Attia, Kamran Khan, and Ralph Wittig. *Deep Learning with INT8 Optimization*

- on Xilinx Devices (White Paper)*. Tech. rep. Xilinx, Inc., 2017. (cit. on p. 414).
- [YY11] Wen Bin Ye and Ya Jun Yu. “Switching Activity Analysis and Power Estimation for Multiple Constant Multiplier Block of FIR Filters”. In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2011, pp. 145–148. (cit. on p. 373).



13

CHAPTER 13

Division by Constants

(...) and putteth other numbers in their place which perform as much as they can do, only by addition and subtraction, division by two, or division by three.

John Napier

This chapter is dedicated to the Euclidean division by an integer constant. Of particular interest are small constants such as 3, 5, etc., for which several specific techniques can be used.

Division by a small integer constant is an operation that occurs often enough to justify investigating a specific operator for it. Of course, divisions by powers of two can be implemented as shifts. Beyond that:

- Euclidean division by 5 occurs in decimal-binary conversions.
- Low-latency division by a small integer constant is needed when interleaving memory banks in numbers that are not powers of two: If we have D memory banks, an address A must be translated to address A/D in bank $A \bmod D$.
- It is also a useful building block for larger operators. For instance, integer division by 3 (with remainder) is needed in the exponent processing for a floating-point cube root.
- In a more contrived example, we will need in Chap. 20 a specific division by 3 to manage the coefficient 1/6 in a Taylor polynomial for the sine function.

Floating-point division by small integers is also very useful. For illustration, the Polybench benchmark suite [Pou12] contains several stencil codes, most of which contain a division by a small constant. The *Jacobi-1d* benchmark contains two divisions by 3; *Jacobi-2d* contains two divisions by 5;

Seidel-2d contains a division by 9; *Fdtd-2d* contains a multiplication by 0.7 that can be transformed to a multiplication by 7 and a division by 10. The management of floating point will be studied in Sect. 15.2.

This chapter is dedicated to the Euclidean division of an integer by a constant, a problem that has been studied quite extensively in a hardware context [AHS76; Li85; Din12; DCC12], with good surveys in [SP94; Dor95; DCC12]. The selected methods presented here are implemented as the `IntConstDiv` operator of FloPoCo.

Section 13.1 presents techniques based on multiplying by the reciprocal. Section 13.2 presents a table-based technique well suited to the microarchitecture of current FPGAs. Section 13.3 then introduces a variation that leads to binary-tree implementations with shorter latency but larger area. All these architectures compute quotient and remainder. Section 13.4 then considers the case when only the quotient, or only the remainder, are needed. Section 13.5 studies architectures composed of two successive divisions in the case when the divider is a product of two small numbers.

All through this chapter, we use the following notations (see also Table 13.1): We compute the quotient Q and remainder R of the Euclidean division of X (a w -bit number) by constant divisor D .

Table 13.1 Notations used in this chapter.

X, D, Q, R	Dividend, divisor, quotient, and remainder such that $X = DQ + R$ with $0 \leq R < D$
k	Size of a chunk of X , i.e., X and Q are considered in radix 2^k
r	Size in bits of $D - 1$
X_i, Q_i	Sub-words (or digits in radix 2^k) of X and Q
w	Number of bits of the dividend X
m	Number of radix- 2^k digits of the dividend X
α	When targeting FPGAs, number of inputs to the architectural LUT

13.1 Multiplying by the Reciprocal

Multiplying by the reciprocal of the constant is a very sensible idea. Indeed the reciprocal is itself a constant; therefore, the range of techniques studied in Chap. 12 can be used here. However, a problem is that the reciprocal, in general, has an infinite (although periodical) binary representation.

The state of the art of reciprocal-based architectures [DCC12] builds upon a software approach [Rob05]. The corresponding architecture is called **Recip**

in the following. Its core idea is to determine three integers A , B , and p such that

$$\left\lfloor \frac{X}{D} \right\rfloor = \left\lfloor \frac{AX + B}{2^p} \right\rfloor \quad \forall X \in \{0, \dots, 2^{w-1}\} \quad (13.1)$$

More specifically, it first determines the minimal bitwidth w_A of A such that (13.1) can be satisfied. There are two possible choices for w_A in [DCC12], and the smallest one is chosen. In FloPoCo, the value of w_A chosen is not always the smallest one, but the one that minimizes the area of the corresponding multiplier by A . Otherwise, this reimplementation of the Recip method is faithful to [DCC12], which indeed provides the minimal value of w_A . One may expect Recip to have a timing complexity of roughly $\log(w)$ and an area complexity of w^2 .

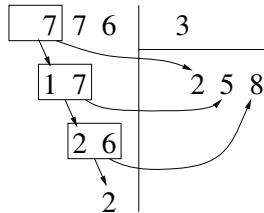
In software, division by a constant is best performed through multiplication by the reciprocal. For instance, although Intel processors have always included division instructions, optimizing compilers for these processors (we tried GCC and CLang) will implement the division by 3 of an unsigned 32-bit integer by multiplication by 0xAAAAAAAB followed by a shift. This magic constant is simply $(2^{33} + 1)/3$, i.e., the best approximation to 1/3 that fits on 32 bits. A similar technique is used for signed integers and even for floating-point numbers [BMR04]. In other words, compilers also use Eq. (13.1), but minimize p among the few values offered by the architecture (8, 16, 32, and 64 bits on a recent Intel processor). For instance, it turns out that the $32 \times 32 \rightarrow 64$ -bit multiplier is adequate for most 32-bit integer divisions.

However, not all processors offer the necessary multiplier. In the AMD/Xilinx MicroBlaze 32-bit soft-core processor, the compiler generates a call to a software integer division routine (tens of cycles), unless invoked with the compiler option `-mwl-multiply-high`. This option requires that the processor is built with the optional instruction to recover the high part of a multiplication. Then, the constant division still requires four instructions (and a few more cycles).

The hardware dividers reviewed in this chapter all have much shorter latency than the software solution. Besides, some of them also provide the remainder of the division for free, or with very little overhead.

13.2 Linear Table-Based Architecture

The table-based technique studied below was introduced in [DD12]. Prior to that, it had, to our knowledge, only been described in lecture notes [Pap06] as an example of combinational circuit. It is in essence a straightforward adaptation of the paper-and-pencil division algorithm in the case of small divisors. Figure 13.1 illustrates this algorithm on the example of the division of the decimal number 776 by 3.



We first compute the Euclidean division of 7 by 3. This gives the first digit of the quotient, here 2, and the remainder is 1. We now have to divide 176 by 3. In the second iteration, we divide 17 by 3: the second quotient digit is 5, and the remainder is 2. The third iteration divides 26 by 3: the third quotient digit is 8, the remainder is 2, and this is also the remainder of the division of 776 by 3.

Fig. 13.1 Illustrative example: division by 3 in decimal.

The key observation is that, in this example, the iteration body consists of the Euclidean division by 3 of a 2-digit decimal number. The first of these two digits is a remainder from previous iteration: Its value is 0, 1, or 2, but no larger. We may therefore implement this iteration with a look-up table (LUT), which, for each value from 00 to 29, gives the quotient and remainder of its division by 3. This small LUT will allow us to divide numbers of arbitrary size by 3.

The reason why this technique was long ignored in the literature is probably that the core of its iteration is itself a (smaller) division: It does not reduce to either additions or multiplications. However, it is relevant in hardware (and even more so on FPGAs) if the LUT is small enough.

13.2.1 Radix- 2^k Representation

Let k be a small integer. We will use the representation of the input dividend X in radix 2^k (the well-known hexadecimal notation is an example of such representation when $k = 4$). The radix- 2^k representation of the binary number X is simply obtained by breaking down the binary representation of X into m chunks of k bits (also see Fig. 13.3):

$$X = \sum_{i=0}^{m-1} X_i \cdot 2^{ki} \quad \text{where} \quad X_i \in \{0, \dots, 2^k - 1\} \quad (13.2)$$

13.2.2 Algorithm

The following algorithm computes the quotient Q and remainder R of the high radix Euclidean division of X by constant D . Each step of this algorithm computes a partial dividend Y_i , a partial remainder R_i , and one radix- 2^k digit of the quotient, Q_i .

Algorithm 13.1: LUT-based computation of X/D

```

function ConstantDiv ( $X, D$ )
   $R_m \leftarrow 0$ 
  for  $i = m - 1$  down to 0 do
     $| Y_i \leftarrow X_i + 2^k R_{i+1}$            // This + is realized as a concatenation
     $| (Q_i, R_i) \leftarrow (\lfloor Y_i/D \rfloor, Y_i \bmod D)$       // read from a table
  end for
  return ( $Q = \sum_{i=0}^{m-1} Q_i \cdot 2^{ki}, \quad R = R_0$ )

```

The line $Y_i \leftarrow X_i + 2^k R_{i+1}$ is simply the concatenation of a remainder and a radix- 2^k digit. This corresponds in Fig. 13.1 to the consideration of one more digit of the dividend.

Let us define r as bitwidth of the largest possible remainder:

$$r = \lceil \log_2(D - 1) \rceil \quad (13.3)$$

Note that r is also the bitwidth of D , as D is not a power of two. Then, Y_i is of size $k + r$ bits. The second line of the loop body, $(Q_i, R_i) \leftarrow (\lfloor Y_i/D \rfloor, Y_i \bmod D)$, computes a radix- 2^k digit and a remainder: It may be implemented as a LUT with $k + r$ bits of input and $k + r$ bits of output (Fig. 13.2).

Theorem 13.1 Algorithm 13.1 computes the Euclidean division of X by D : It outputs the quotient Q and the remainder R so that $X = Q \times D + R$ and $R < D$. The radix- 2^k representation of the quotient Q is also a binary representation, each iteration producing k bits of this quotient.

Proof The proof proceeds in two steps. First, Lemma 13.1 states that $X = D \times \sum_{i=0}^m Q_i \cdot 2^{-ki} + R_0$. This shows that we compute some kind of Euclidean division, but it is not enough: We also need to show that the Q_i form a binary representation of the result. For this, it is enough to show that they are radix- 2^k digits, which is established through Lemma 13.2.

Lemma 13.1

$$X = D \sum_{i=0}^m Q_i \cdot 2^{-ki} + R_0$$

Proof By definition of Q_i and R_i we have $Y_i = DQ_i + R_i$.

$$\begin{aligned} X &= \sum_{i=0}^{m-1} X_i \cdot 2^{-ki} \\ &= \sum_{i=0}^{m-1} (X_i + 2^k R_{i+1}) \cdot 2^{-ki} - \sum_{i=0}^{m-1} (2^k R_{i+1}) \cdot 2^{-ki} \\ &= \sum_{i=0}^{m-1} (DQ_i + R_i) \cdot 2^{-ki} - \sum_{i=1}^m R_i \cdot 2^{-ki} \\ &= D \sum_{i=0}^{m-1} Q_i \cdot 2^{-ki} + R_0 - R_m \cdot 2^{-km} \end{aligned}$$

and $R_m = 0$.

Lemma 13.2 $\forall i \quad 0 \leq Y_i \leq 2^k D - 1$

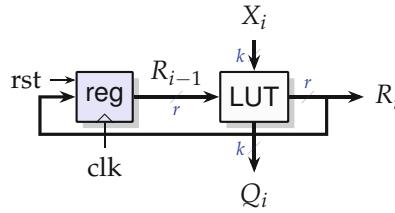


Fig. 13.2 Sequential architecture for Algorithm 13.1: LUT-based division of a number written in radix- 2^k by a constant.

Proof The digit X_i verifies by definition $0 \leq X_i \leq 2^k - 1$; R_{i+1} is either 0 (initialization) or the remainder of a division by D ; therefore, $0 \leq R_i \leq D - 1$. Therefore, $Y_i = X_i + 2^k R_{i+1}$ verifies $0 \leq Y_i \leq 2^k - 1 + 2^k(D - 1)$, or $0 \leq Y_i \leq 2^k D - 1$.

We deduce from the previous lemma and the definition of Q_i as quotient of Y_i by D that

$$\forall i \quad 0 \leq Q_i \leq 2^k - 1$$

which shows that the Q_i are indeed radix- 2^k digits. Thanks to Lemma 13.1, they are the digits of the quotient.

The algorithm computes k bits of the quotient in each iteration: The larger k is, the fewer iterations are needed for a given input number with bitwidth w . However, the larger k is, the larger the required LUT. Section 13.2.4 quantifies this trade-off.

13.2.3 Iterative or Unrolled Implementation of the Basic Recurrence

The iteration may be implemented sequentially as depicted in Fig. 13.2 or as the fully unrolled architecture depicted in Fig. 13.3. In all of the following, we will focus on the latter (called LinArch, short for linear architecture) because it enables high-throughput pipelined implementations.

13.2.4 Cost Evaluation of LinArch

Algorithm 13.1 performs $m = \lceil w/k \rceil$ iterations. For a given D and k , the architecture therefore grows linearly with the input size w . Note that general division or multiplication architectures grow quadratically.

Let us now consider how it grows with the constant D .

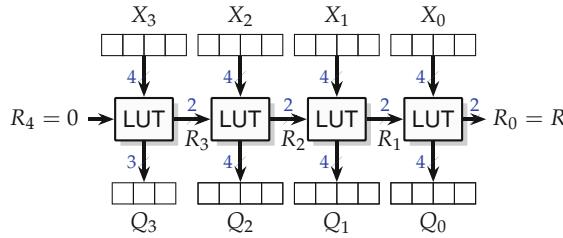


Fig. 13.3 Unrolled architecture (LinArch) for Algorithm 13.1: LUT-based division by 3 of a 16-bit number written in radix 2^4 ($k = 4, r = 2$).

The area of a LUT in ASIC is essentially proportional to the number of bits it holds. One LUT of Figs. 13.2 and 13.3 needs to store $D \cdot 2^k$ entries of $r + k$ bits, or $D \cdot 2^k(r + k)$ bits. To remove D from this formula, we may round up the table size to a power of two: One table will store $2^{r+k}(r + k)$ bits, some of which are “don’t care” values. The latter can be optimized out by synthesis tools. Table 13.2 justifies removing D from the equation: It shows that the tools are able to considerably reduce the area compared to the predicted area. However, different D (5 and 7) with the same r and k lead to exactly the same area and delay results.

Finally, the area cost of the sequential architecture grows as $2^{r+k}(r + k)$ and that of LinArch as $\lceil w/k \rceil 2^{r+k}(r + k)$.

The delay of a LUT is essentially proportional to the number of input bits. For both architectures, the delay grows as $\lceil w/k \rceil(r + k)$ – it is linear in w (this was obvious in Fig. 13.3).

Section 13.3 will introduce a parallel architecture that offers smaller delay, sometimes at the expense of larger area.

Hands on: Integer constant division using LinArch

The following FloPoCo call generates VHDL code for the Euclidean division of a 32-bit integer by 3, using the LinArch architecture.

```
flopoco IntConstDiv wIn=32 d=3
```

You may want to add the frequency=10 option to obtain a combinatorial (non-pipelined) architecture.

Observe that the number of LUT instances grows when moving to d=3 and then to d=17.

13.2.5 FPGA-Specific Remarks

The basic reconfigurable logic block of current FPGAs is a small LUT with α inputs and one output (see Chap. 4) with α typically between 4 and 6. It is denoted $\text{LUT}\alpha$ in this book. These $\text{LUT}\alpha$ can also be used to build the $(r+k)$ -input, $(r+k)$ -output LUTs needed for constant division architectures.

Hence, on field programmable gate arrays (FPGAs), there is no point in using tables with fewer than α inputs. The value of k should therefore be chosen such that $r+k \geq \alpha$. Then, the cost in terms of $\text{LUT}\alpha$ of an $(r+k)$ -input, $(r+k)$ -output LUT is no longer $2^{r+k}(r+k)$ but $2^{r+k-\alpha}(r+k)$. When $r+k > \alpha$, this formula is a first approximation: It assumes that the FPGA architecture includes multiplexers for assembling smaller $\text{LUT}\alpha$ into larger LUTs, which is true, on all architectures, up to a certain value of $r+k-\alpha$. Beyond this value, additional $\text{LUT}\alpha$ will be used for such multiplexing.

With this approximation, the area of LinArch in $\text{LUT}\alpha$ is $\lceil w/k \rceil 2^{\max(r+k-\alpha, 0)}(r+k)$.

Therefore, the optimal choice of k in terms of area is the smallest k such that $r+k \geq \alpha$. As $r = \lceil \log_2(D-1) \rceil$, the method is very area-efficient for small values of D .

As discussed in detail in Chap. 4, in each FPGA family, there are restrictions on LUT utilization that allow either several smaller look-up tables (LUTs) per basic logic element (BLE) with typically 4 to 5 inputs or a single large LUT per BLE with typically 6 inputs.

We may use, for instance, 6-input LUTs to implement division by 3 ($r=2$) in radix 16 ($k=4$), as illustrated by Fig. 13.3. Implementing the core loop (i.e., the sequential part of Fig. 13.2) costs just 6 LUTs (for a 6 bits in, 6 bits out table). The cost for the complete LinArch for w bits is $m = \lceil w/4 \rceil \times 6$ LUT6, for instance, 36 LUT6 for 24 bits (single precision), or 78 LUTs for 53 bits (double precision).

Table 13.2 reports some synthesis results. This table illustrates that the method is mostly suited to small constants: For this FPGA ($\alpha=6$), starting with $D=17$, we have $k=1$, so the architecture requires as many LUTs as there are bits in the input. Besides, the size of these LUTs then grows as the exponential term $2^{r+k-\alpha}$.

13.3 Parallel Division

The family of binary tree constant division (BTCD) circuits has been introduced in [Ugū+16] and studied in more detail in [Ugū+17]. BTCD is to LinArch what fast adders are to carry-propagate adders: Where latency and area of LinArch are linear in w , the latency of BTCD is logarithmic in w , at the expense of an area that is proportional to $w \log(w)$.

Table 13.2 LinArch dividers of a 32-bit value by D (evaluated on Kintex-7).

D	r	k	Estimation	Synthesis results	
				Area (LUT6)	Area (LUT)
3	2	4	48	32	6.0
5	3	3	66	45	9.3
7	3	3	66	45	9.3
9	4	2	96	87	17.9
11	4	2	96	87	17.9
17	5	1	192	165	18.5

This technique has been implemented in FloPoCo. We refer the reader to [Ugū+17] for details and comparisons of the three methods LinArch, BTCD, and Recip.

Hands on: Integer constant division using BTCD

The following FloPoCo call generates VHDL code for the Euclidean division of a 32-bit integer by 3, using the BTCD architecture.

```
flopoco IntConstDiv arch=1 wIn=32 d=3
```

Another option is `arch=2` for an architecture based on a constant multiplication by the reciprocal.

13.4 Remainder-Only or Quotient-Only

This section discusses some variants of the previous architectures that are more efficient by only outputting the remainder or only outputting the quotient.

13.4.1 Reciprocal Method Outputting Only the Quotient

The reciprocal method as published in [DCC12] only computes the quotient. The previous tables report results for architectures that are slightly more powerful, as they also compute the remainder as $R = X - DQ$. However, the overhead of this computation is very small, as Table 13.3 shows. Indeed, this computation has to be only performed on the number of bits r of R ,

which is small with respect to the number of bits of X . All the other bits of the subtraction can be predicted to be zeroes.

Table 13.3 The overhead of computing the remainder in the Recip method, evaluated on Kintex-7 FPGA. Recip/R denotes the architecture of [DCC12] that computes quotient only.

D	w	Recip/R		Recip	
		Area Delay (LUT)	Delay (ns)	Area Delay (LUT)	Delay (ns)
3	8	17	3.6	16	3.5
	16	52	4.5	51	4.2
	32	139	6.1	138	6.0
	64	346	8.5	345	7.9
	128	825	12.4	824	13.3
23	8	26	4.0	20	3.7
	16	83	5.1	71	3.8
	32	169	7.3	158	5.8
	64	401	9.0	390	7.9
	128	931	13.4	920	14.0

13.4.2 Remainder-Only Variant of LinArch and BTCD

In LinArch, if one only needs the remainder, the quotient bits need not be stored at all in the LUTs. This entails savings in terms of area that are easy to predict: roughly a factor $r/(r+k)$, as illustrated by Table 13.4. However, there is almost no improvement in delay, as the critical path is unchanged (see Fig. 13.3).

In BTCD, computing only the remainder improves both area and delay. The architecture becomes a binary tree of tables that have $2r$ inputs and r outputs.

As Table 13.4 shows, BTCD therefore offers a much better area-time trade-off than LinArch if only the remainder is needed. For instance, for $D = 5$, the area is nearly identical (both architectures build the same number of LUTs with 6 inputs and 3 outputs), while the delay of BTCD is logarithmic instead of linear for LinArch.

Table 13.4 Comparison of remainder-only architectures (R) to Euclidean divider architectures (Q+R), evaluated on Kintex-7 FPGA.

D	w	LinArch				BTCD			
		Q+R		R		Q+R		R	
		Area (LUT)	Delay (ns)	Area (LUT)	Delay (ns)	Area (LUT)	Delay (ns)	Area (LUT)	Delay (ns)
3	8	8	3.7	3	3.6	12	3.6	2	3.6
	16	15	3.8	5	3.5	37	3.7	5	3.7
	32	32	6.0	11	6.5	95	4.8	12	3.8
	64	63	13.5	21	14.2	225	6.2	25	4.8
	128	128	26.3	43	27.2	517	8.4	50	5.3
5	8	9	3.6	6	3.6	18	3.6	7	3.5
	16	21	4.4	14	4.4	44	3.8	14	3.6
	32	45	9.3	30	9.8	109	4.7	29	3.7
	64	93	20.1	62	21.2	270	6.7	62	4.5
	128	189	38.3	126	37.7	612	9.0	129	5.2

Hands on: Computing the remainder only

The following FloPoCo call generates VHDL code for computing the remainder of the Euclidean division of a 32-bit integer by 3.

```
flopoco IntConstDiv computeQuotient=false wIn=32 d=3
```

13.5 Composite Division

The table-based methods scale poorly with the size (in bits) of the constant. For a large constant that is the product of two smaller ones, it is however possible to divide successively by the two smaller constants. This will often lead to a smaller architecture than a monolithic division by the large constant. The following first formalizes this intuition and then discusses the choice of an optimal factorization.

13.5.1 Algorithm

Let us first assume that D is the product of two smaller constants:

$$D = D_a \times D_b \quad (13.4)$$

The Euclidean division of X by D_a can be written

$$X = D_a Q_a + R_a \quad \text{with } 0 \leq R_a \leq D_a - 1. \quad (13.5)$$

Then we can divide Q_a by D_b :

$$Q_a = D_b Q_b + R_b \quad \text{with } 0 \leq R_b \leq D_b - 1. \quad (13.6)$$

Putting all together, we obtain

$$\begin{aligned} X &= D_a D_b Q + D_a R_b + R_a \\ &= D \times Q + R \quad \text{where } R = D_a R_b + R_a. \end{aligned} \quad (13.7)$$

Since $R = D_a R_b + R_a \leq D_a(D_b - 1) + D_a - 1 = D - 1$, R is indeed the remainder of the division of X by D , and Q_b is indeed the proper quotient.

Now, if the divisor D is the product of more than two factors (i. e., if D_a or D_b can themselves be factored), the previous decomposition can be applied recursively.

13.5.2 Architecture

The architecture (shown in Fig. 13.4) consists in two of the previous dividers to compute the Euclidean divisions by D_a and D_b , plus a multiply-and-add to compute $R = D_a R_b + R_a$.

This additional multiply-and-add is typically small for two reasons. First, the remainders are small. Second, this is a constant multiplication, for which, again, we can use the range of techniques studied in Chap. 12.

As can be seen in Fig. 13.4, the critical path is that of the divider with the largest m , plus one LUT for the quotient output, and a small constant multiplication and addition for the remainder output.

13.5.3 Results and Discussions

Table 13.5 provides synthesis results on FPGA for three constants that are of practical significance: 9 appears in some 2D stencils, 15 is (up to a power of two) 60 and occurs in timing conversions, and 45 is (up to a power of two) 360 which appears in degree/radian conversions. Interestingly, the two latter constants were probably chosen in ancient times because they can be divided by 2, 3, 4, and 5.

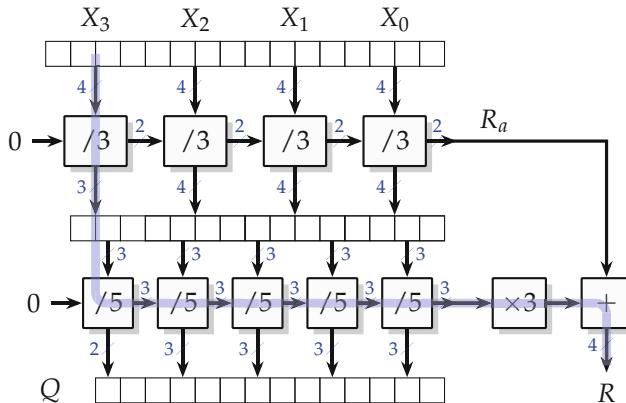


Fig. 13.4 Composite divider of a 16-bit value by $15 = 3 \times 5$, with the critical path highlighted.

Table 13.5 shows that composite dividers are not only smaller; they are also consistently faster than the atomic LinArch on FPGAs. This is easily explained on the example of division by 9. The atomic LinArch divider by 9 is predicted to require about the same area as the composite divider. However, its optimal value of k is 2, and the 32-bit input X is therefore decomposed in $m = 16$ radix- 2^2 digits: The architecture has 16 LUTs on the critical path. When we compose two dividers by 3, the optimal value of k is 4 for each, so the architecture of each subdivider has only $m = 8$ LUTs on the critical path. As Fig. 13.4 shows, the critical path of the composite architecture is therefore 8 + 1 LUTs only (plus the remainder reconstruction) instead of 16 LUTs.

Hands on: Computing division by a product of small constants

The following FloPoCo call generates VHDL code for computing the remainder of the Euclidean division of a 32-bit integer by 15 using successive divisions by 3 and 5.

```
flopoco IntConstDiv arch=1 computeQuotient=false \
wIn=32 d=3:5
```

A floating-point divider by a small integer constant, based on the architectures of this chapter, will be described in Sect. 15.2.

Table 13.5 Examples of composite 32-bit dividers on Kintex7. Estimates ignore the cost of the remainder reconstruction, which is accounted for in the measured area and delay. In all cases the optimal value of k is used.

D	Decomposition	Predicted	Measured	
		Area (LUT)	Area (LUT)	Delay (ns)
9	Atomic Recip	–	184	6.2
	Atomic BTCD	–	218	6.1
	Atomic LinArch	96	87	17.9
	3×3	$48+48=96$	65	7.0
15	Atomic Recip	–	97	6.2
	Atomic BTCD	–	199	5.8
	Atomic LinArch	96	87	17.9
	5×3	$66+48=114$	80	9.6
45	3×5	$68+46=114$	75	9.4
	Atomic Recip	–	207	8.0
	Atomic LinArch	448	371	24.3
	9×5	$96+60=156$	134	18.4
45	5×9	$66+90=156$	134	18.3
	$5 \times 3 \times 3$	$66+48+48=162$	108	10.0
	$3 \times 3 \times 5$	$48+48+60=156$	113	9.8
	15×3	$96+48=144$	120	18.5
	3×15	$48+96=144$	125	17.6

References

- [AHS76] Ehud Artzy, James A. Hinds, and Harry J. Saal. “A Fast Division Technique for Constant Divisors”. In: *Communications of the ACM* 19 (1976), pp. 98–101. (cit. on p. [428](#)).
- [BMR04] Nicolas Brisebarre, Jean-Michel Muller, and Saurabh Kumar Raina. “Accelerating Correctly Rounded Floating-Point Division when the Divisor Is Known in Advance”. In: *IEEE Transactions on Computers* 53.8 (2004), pp. 1069–1072. (cit. on p. [429](#)).
- [DCC12] Theo Drane, Wai-Chuen Cheung, and George Constantinides. “Correctly Rounded Constant Integer Division via Multiply-Add”. In: *International Symposium on Circuits and Systems (ISCAS)*. 2012, pp. 1243–1246. (cit. on pp. [428](#), [429](#), [435](#), [436](#)).

- [DD12] Florent de Dinechin and Laurent-Stéphane Didier. "Table-Based Division by Small Integer Constants". In: *Applied Reconfigurable Computing*. Vol. 7199. LNCS. Springer, 2012, pp. 53–63. (cit. on p. 429).
- [Din12] Florent de Dinechin. "Multiplication by Rational Constants". In: *IEEE Transactions on Circuits and Systems, II* 59.2 (2012), pp. 98–102. (cit. on p. 428).
- [Dor95] R. W. Doran. "Special Cases of Division". In: *Journal of Universal Computer Science* 1.3 (1995), pp. 67–82. (cit. on p. 428).
- [Li85] Shuo-Yen Robert Li. "Fast Constant Division Routines". In: *IEEE Transactions on Computers* C-34.9 (1985), pp. 866–869. (cit. on p. 428).
- [Pap06] Andrew Paplinski. *CSE2306/1308 Digital Logic Lecture Note, Lecture 8*. Clayton School of Information Technology Monash University, Australia. 2006. (cit. on p. 429).
- [Pou12] Louis-Noël Pouchet. *Polybench: The polyhedral benchmark suite*. <https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/>. 2012. (cit. on p. 427).
- [Rob05] Arch D. Robison. "N-Bit Unsigned Division Via N-Bit Multiply-Add". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2005, pp. 131–139. (cit. on p. 428).
- [SP94] P. Srinivasan and F.E. Petry. "Constant-division algorithms". In: *IEE Proc. Computers and Digital Techniques* 141.6 (1994), pp. 334–340. (cit. on p. 428).
- [Ugū+16] H. Fatih Uğurdağ, Anil Bayram, V. Emre Levent, and Sezer Gören. "Efficient Combinational Circuits for Division by Small Integer Constants". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2016, pp. 1–7. (cit. on p. 434).
- [Ugū+17] H. Fatih Uğurdağ, Florent de Dinechin, Y. Serhan Gener, Sezer Gören, and Laurent-Stéphane Didier. "Hardware division by small integer constants". In: *IEEE Transactions on Computers* 66.12 (2017), pp. 2097–2110. (cit. on pp. 434, 435).



14

CHAPTER 14

Fixed-Point Squares, Cubes, and Other Integer Powers

A squarer is another useful specialization of multiplication in the case when the two inputs are identical. It is useful in all sorts of coarser components such as Euclidean norms or polynomials. Besides, operators for X^p for $p > 2$ can also be built efficiently out of squares and multiplications.

This chapter addresses the computation of X^p for a positive integer p . It presents another specialization of multiplication and therefore builds upon Chaps. 7 and 8.

X^{-p} (reciprocal square, cubes, etc.) and $X^{\frac{1}{p}}$ (square root, cube root, etc.) are algebraic functions that will be studied in Chap. 19.

Note that all these functions are also numerical functions of one variable and as such can benefit from all the approximation methods studied in Part III of this book. These techniques will not be covered in this chapter, but the interested reader should keep them in mind. In particular, for very low precisions (8 bits and below), plain tabulation of these functions is a very attractive option.

14.1 Generalities

For a positive integer p and a fixed-point input $X \in \text{ufix}(m, \ell)$, the largest possible value is $2^{m+1} - 2^\ell$ and the smallest nonzero value is 2^ℓ . Therefore, the exact value of X^p fits in a $\text{ufix}(p(m+1), p\ell)$ format.

For a signed fixed-point input $X \in \text{ufix}(m, \ell)$, we have $X \in [-2^m, 2^m - 2^\ell]$, and the smallest nonzero value is again 2^ℓ . The LSB of the format to use for X^p will always be $p\ell$ as in the unsigned case. However, the signedness of the format to use for X^p depends on the parity of p .

- If p is even, then X^p will always be positive. The largest possible value of X^p is 2^{mp} . Therefore, the format to use for X^p is $\text{ufix}(pm, p\ell)$.
- If p is odd, then X^p will have the sign of X , and $X^p \in [-2^{mp}, (2^m - 2^\ell)^p]$. The format to use for X^p is $\text{sfix}(pm, p\ell)$.

All the previous concerns the exact value X^p , but many applications will accommodate truncated values whose least significant bit (LSB) is higher than that of the exact value.

14.2 Squarers

As for the multipliers, we focus here on the square of an integer X , but all the work presented here can trivially be transposed to any fixed-point format for X .

14.2.1 Basics

The bit complexity of squaring is roughly half of that of standard multiplication. To understand why, Fig. 14.1 shows the tiling board (as introduced in Sect. 8.4.2) of a full radix-2 multiplier computing $X \times X$. There, each partial product on the diagonal is computing $x_i x_i = x_i$. Each partial product $x_i x_j$ above the diagonal (shown in red) has a symmetrical $x_j x_i$ below the diagonal. As these two partial products have the same weight 2^{i+j} , we may compute $x_i x_j$ only once and double its value by giving it the weight 2^{i+j+1} .

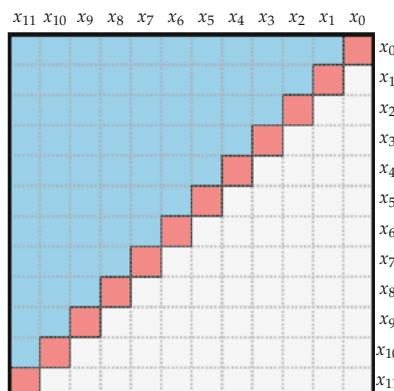


Fig. 14.1 Tile view of the computation of a 12-bit square. Diagonal partial products $x_i x_i = x_i$ are in red, and bits $x_i x_j$ such that $i > j$ are in blue.

Another point of view is that the bit-level definition of a square can be written

$$X^2 = \left(\sum_{i=0}^{n-1} 2^i x_i \right)^2 = \sum_{i=0}^{n-1} 2^{2i} x_i + \sum_{0 < j < i < n} 2^{i+j+1} x_i x_j . \quad (14.1)$$

Figure 14.2 is an example of squarer bit heap obtained using the previous formula.

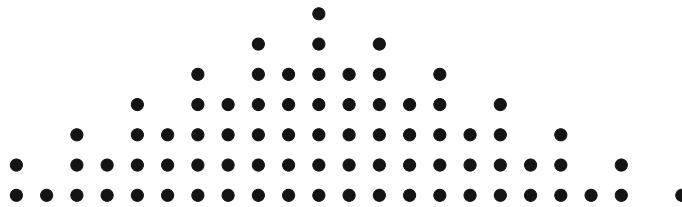


Fig. 14.2 Bit heap of an exact 12-bit squarer.

14.2.2 High-Radix Square

A similar property holds for a splitting of the input into several subwords of k bits (or considering for X a radix- 2^k representation):

$$X^2 = (2^k X_1 + X_0)^2 = 2^{2k} X_1^2 + 2 \cdot 2^k X_1 X_0 + X_0^2 \quad (14.2)$$

$$\begin{aligned} (2^{2k} X_2 + 2^k X_1 + X_0)^2 &= 2^{4k} X_2^2 + 2^{2k} X_1^2 + X_0^2 \\ &\quad + 2 \cdot 2^{3k} X_2 X_1 \\ &\quad + 2 \cdot 2^{2k} X_2 X_0 \\ &\quad + 2^k X_1 X_0 \end{aligned} \quad (14.3)$$

Each square or product of the above equation may be computed by a multiplier (e.g., a DSP block in FPGAs). Therefore, (14.2) and (14.3) yield a reduction of the DSP count from 4 to 3, or from 9 to 6 for a 2-way or 3-way splitting of input X , respectively. Besides, this reduction comes at no arithmetic overhead (contrary to Karatsuba where it costs a few extra additions).

However, an issue arises if we want to exploit the DSP chaining capabilities introduced in Sect. 8.4.2.4, p. 231. In AMD devices, for instance, the DSP chain input is only able to perform a shift by 17 bits. In (14.2) with $k = 17$, the powers are 0, 18, and 34. In (14.3) they are 0, 18, 34, 35, 42, 64.

To address this, one more trick may be used for integers of at most 33 bits. Equation (14.2) is rewritten

$$(2^{17}X_1 + X_0)^2 = 2^{34}X_1^2 + 2^{17}(2X_1)X_0 + X_0^2 \quad (14.4)$$

and $2X_1$ is computed by shifting X_1 by one bit before inputting it to the corresponding DSP. We have this spare bit only if the size of X_1 is at most 16, i. e., if the size of X is at most 33. This technique may therefore be used in the important case of 32-bit integers.

This trick applies similarly to Intel FPGAs, albeit with different values of the input shift. For instance, the cascading inputs on Stratix V DSP block support 18-bit and 27-bit shifts.

14.2.3 Booth Recoding

Booth recoding is a special case of radix-4 representation that was introduced in detail in Sect. 8.3.2, p. 215, where it almost divided by two the size of a multiplier bit heap. It is therefore natural to consider it for squarers as well. Booth recoding consists in rewriting X as

$$X = \sum_i 4^i Z_i \quad \text{with} \quad Z_i \in \{-2, -1, 0, 1, 2\}. \quad (14.5)$$

Then

$$X^2 = \sum_i 4^{2i} Z_i^2 + \sum_{i>j} 4^{i+j+1} Z_i Z_j. \quad (14.6)$$

As each Z_i is a 2-bit number, each radix-4 product $Z_i Z_j$ replaces a small square of 4 binary partial products $x_i x_j$ in (14.1). Now, $Z_i^2 \in \{0, 1, 4\}$ and can be written as two bits in a bit heap, replacing in a radix-2 squarer 3 bits (two red bits in Fig. 14.1, x_i^2 and x_{i+1}^2 , and a blue one $x_i x_{i+1}$). For the bulk of the computation, $Z_i Z_j \in \{-4, -2, -1, 0, 1, 2, 4\}$ and can be written as 3 sign-extended bits in a bit heap. These 3 bits replace a square of 4 blue bits of the radix-2 case. The expected reduction factor in bit heap size is therefore about 3/4.

Considering the overhead of Booth recoding itself and the cost of the Boolean function that must compute $Z_i Z_j$ (to be compared with 4 AND gates in the plain approach), a squarer using Booth recoding is probably less efficient than a simpler squarer using (14.1).

14.2.4 Truncated Squarers

There is nothing special about the construction of truncated squarers; the techniques introduced in Sect. 7.2.4, p. 167, can be used directly. As for mul-

tipliers, it is possible to refine this technique to improve the statistical distribution of the error [Gar+10], but the generic bit heap truncation heuristic is optimal with respect to worst-case error.

As an illustration, Fig. 14.3 shows the bit heap of a faithful squarer inputting and outputting numbers in $[0, 1)$ in the ufix($-1, -12$) format. This bit heap is obtained by truncating the exact squarer depicted in Fig. 14.2 to be faithful to its 12 most significand bits, using Algorithm 7.1, p. 172. In this example, $\ell_{\text{ext}} = \ell - 3$ and $t = 3$, and the bit heap includes 3 constant ones in its three rightmost columns.

As a rule of thumb, it is possible to truncate one column more from a squarer than from the corresponding multiplier: Each truncated column in the squarer is half the size of the column at the same position in the multiplier; therefore, its truncation contributes half the error.

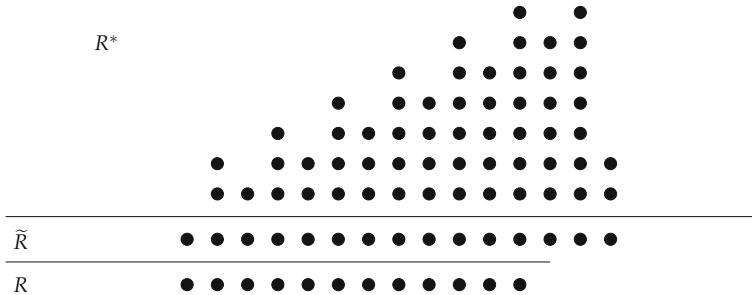


Fig. 14.3 Bit heap of a faithful truncated 12-bit squarer.

This example was built using the algebraic formulation of (14.1) and is relevant for ASIC implementations. The same truncation technique can also be used on bit heaps produced out of heterogeneous multiplier tiles, which we review now.

14.2.5 Squarer-Specific Tiling for FPGAs

The radix- 2^k decomposition of X is a particular case of tiling where all tiles are square. However, it is also possible to use rectangular and other non-square tiles [BKD22]. Figure 14.4 shows various interesting tile configurations:

- Tile M0 is business as usual. It uses 6 different input bits from X : x_0, x_1, x_9, x_{10} , and x_{11} , so no bit is shared between the horizontal and vertical axes. It is a normal rectangular multiplier tile that we import for our library of multiplier tiles. The only difference, compared to a multiplier, is that

it realizes both M0 tiles shown in Fig. 14.4 by using only one multiplier whose output is shifted left by 1 bit.

- Tile M1 is also a rectangular tile, but it is placed in such a way that it covers diagonal bits. A negative consequence is that a square of bits (noted S1) is computed twice, since S1 is covered both by M1 and by its mirror image. For a correct result, S1 must be recomputed and subtracted from the bit heap. This may be globally beneficial if M1 (or a similar, larger tile) is computed by a DSP block that otherwise efficiently covers a large area.
- The fact that M1 shares two bits (x_7 and x_8) between the horizontal and vertical axes also means that it inputs fewer bits: On our example M1 is a function of 6 inputs only. It could thus be tabulated efficiently in LUT6. But then, why not compute properly also the bottom-left square that remains uncovered on Fig. 14.4 and the diagonal bit x_6 ? And why not manage the overlap situation within this larger tile?
- Tile T1 does exactly that: It is a squarer-specific tile that includes some diagonal bits and some non-diagonal ones, constructed in such a way that it maps very efficiently on LUT6 which are fracturable into two LUT5. It covers 15 partial products of the upper left area (blue + red in Fig. 14.4). It adds 10 bits to the bit heap, using 10 LUT5 grouped as 5 LUT6. Its efficiency is thus $15/(5 + 10 \times .55) \approx 1.43$, which is better than the LUT-based multipliers of Table 8.3, p. 229.
- Note that T1 could also be drawn as a square covering the upper-right corner of Fig. 14.4. Such a square can be computed by a DSP block, but only used in a square configuration ($w_X = w_Y$).

Figure 14.5 provides an example for a 53-bit squarer as required for a floating-point double-precision squarer. All the cases discussed above appear on this example. Note in particular the upper-right square tile, similar to T1 but here computed with a DSP block. This example also illustrates that this combinatorial optimization problem is far from being trivial.

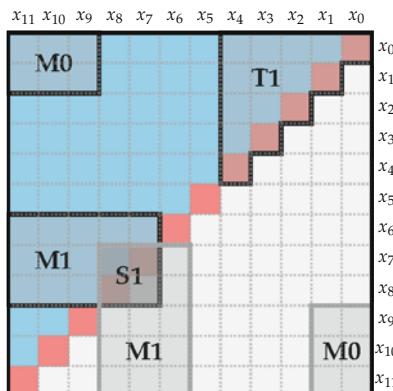


Fig. 14.4 Examples of tiles in a squarer.

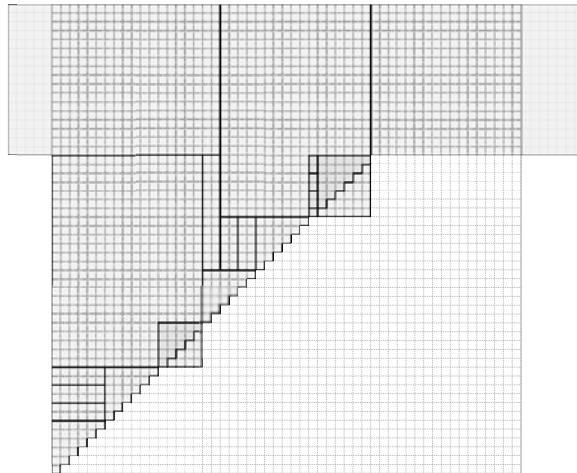


Fig. 14.5 Optimal tiling of a 53-bit squarer with 4 DSP.

Once these squarer-specific tiles have been defined, it is possible to extend the multiplier tiling ILP formulation of Chap. 8 to squarers, both for exact and truncated squarers. For that, only the upper left area has to be covered by tiles (blue + red in Fig. 14.4) with a special treatment of squarer-specific tiles (those are only allowed along the diagonal). The interested reader will find details in [BKD22].

Hands on: Squarers in FloPoCo

The following command produces a plain squarer.

```
flopoco IntSquarer win=8
```

For a tiling-based squarer, the `method=optimal` optional parameter should be specified, and it requires that FloPoCo is linked to a good ILP solver.

14.3 Cubes and Higher Powers

14.3.1 Direct Algebraic Expression

We have already observed cubes as an example of bit heaps in Sect. 7.2.3, p. 163, but they were divided by 3 there.

The binary expansion of a cube is

$$X^3 = \left(\sum_{i=\ell}^m 2^i x_i \right)^3 \quad (14.7)$$

$$\begin{aligned} &= \sum_{\ell \leq i \leq m} 2^{3i} x_i \\ &+ \sum_{\ell \leq i < j \leq m} 3 \cdot (2^{i+2j} + 2^{2i+j}) x_i x_j \\ &+ \sum_{\ell \leq i < j < k \leq m} 3 \cdot 2^{i+j+k+1} x_i x_j x_k. \end{aligned} \quad (14.8)$$

The multiplication by 3 of a bit actually replicates it in the bit heap ($3x = 2x + x$): This formulation is not very interesting. For exact cubes, the number of bits in the bit heap grows as the cube of the input size.

Now, if the application requires a truncated cube, then the truncation techniques of Sect. 7.2.4 may dramatically reduce the bit heap size, as illustrated by Fig. 7.18, p. 173. This must be evaluated in an application-dependent way, as it really depends on the number of bits of the cube which are actually needed by the application.

For higher powers, it becomes even less interesting to expand the binary representation of X^p . For instance, X^4 is probably more efficiently evaluated as $(X^2)^2$, which may use two squarers.

This is a specific case of the generic technique that we introduce now.

14.3.2 Computing Powers by Squaring and Multiplying

Here is a general recursive technique to evaluate X^p :

- If p is even, $p = 2q$, evaluate X^p as $(X^q)^2$.
- If p is odd, $p = 2q + 1$, evaluate X^p as $X \cdot (X^q)^2$.

For example,

$$X^{34} = (X^{17})^2 = (X \cdot X^{16})^2 = \left(X \cdot (X^8)^2 \right)^2 \quad (14.9)$$

$$= \dots = \left(X \cdot \left(\left((X^2)^2 \right)^2 \right)^2 \right)^2. \quad (14.10)$$

This technique costs $\lfloor \log_2 p \rfloor$ squaring operations and $k - 1$ multiplications, where k is the number of ones in the binary representation of p .

Note that for an exact final result, each squarer outputs twice as many bits as it inputs. This solution can become quite expensive.

14.3.3 Approximate Fixed-Point Powers

When the input X is provided in a fixed-point format $\text{ufix}(-1, \ell)$ or $\text{sfix}(0, \ell)$, then $|X| < 1$; hence, $|X^p| < 1$. It is common that the output needs to be truncated or rounded, in particular when it uses the same format.

Correct rounding requires an exact computation of X^P . A faithful result can be achieved at a much lower cost. Using the square and multiply technique, it is indeed possible in this case to limit the size of intermediate computations to the size of the output, plus a few guard bits, using only truncated squarers and multipliers. We do not provide a detailed error analysis here. Roughly speaking the architecture needs one guard bit per squaring or multiplication operations.

Another option [PBM01] is to use the generic approximation methods studied in Part III of this book. In particular, plain tabulation is attractive for very small precisions since it also provides correct rounding.

A floating-point X^k is studied in Chap. 15.

References

- [BKD22] Andreas Böttcher, Martin Kumm, and Florent de Dinechin. “Resource Optimal Squarers for FPGAs”. In: *International Conference on Field Programmable Logic and Application (FPL)*. IEEE, 2022. (cit. on pp. 447, 449).
- [Gar+10] Valeria Garofalo, Marino Coppola, Davide De Caro, Ettore Napoli, Nicola Petra, and Antonio G.M. Strollo. “A novel truncated squarer with linear compensation function”. In: *International Symposium on Circuits and Systems*. IEEE. 2010, pp. 4157–4160. (cit. on p. 447).
- [PBM01] José-Alejandro Piñeiro, Javier D. Bruguera, and Jean-Michel Muller. “Faithful Powering Computation Using Table Look-Up and a Fused Accumulation Tree”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2001, pp. 40–47. (cit. on p. 451).



15

CHAPTER 15

Specialization and Fusion of Floating-Point Operators

This chapter studies the possible optimizations that arise in the specialization or fusion of floating-point operators. It builds upon the specialized fixed operators of previous chapters and focuses on exponent management and rounding issues specific to floating point.

This chapter covers many application-specific floating-point operators without pretending to be exhaustive. It addresses floating-point multiplication by a constant in Sect. 15.1; division by a small integer constant in Sect. 15.2; squares, cubes, etc. in Sect. 15.3; addition of positive terms in Sect. 15.4; combined sum and difference in Sect. 15.5; fused sums of products or squares in Sect. 15.6; and finally in Sect. 15.7 other compiler optimizations that can be performed in an high-level synthesis (HLS) context.

15.1 Floating-Point Constant Multiplication

The purpose of this section is the construction of the operator represented in Fig. 15.1. Here the value of the constant will be denoted as C , and it may be any real number: $C \in \mathbb{R}$. Thus, C may be a floating-point number, but it may also have an infinite binary representation (e.g., $C = 1/3$), or even be irrational (e.g., $C = \pi$). Of course, this section builds upon the constant multipliers of Chap. 12, but it also introduces several optimizations specific to floating point, related to exponent processing and normalization.

This section uses the notations of Sect. 2.5 and the same format for the input X and the output R (beware that some of the architectural tricks in the following is no longer valid if X and R have different values of w_E or w_F).

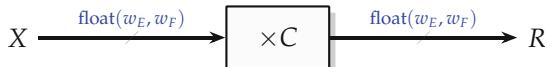


Fig. 15.1 Black-box interface to a floating-point constant multiplier.

For the constant, we define its unique floating-point¹ representation

$$C = (-1)^{s_C} \cdot 2^{E_C} \cdot (1 + F_C) \quad (15.1)$$

such that $F_C \in [0, 1)$. Again, F_C may have an infinite binary representation.

As a starting point, a constant floating-point multiplier may essentially be constructed as a simplification of the standard FP multiplier of Fig. 11.7, p. 344: The significand multiplication is replaced with a constant multiplication, and the exponent addition is replaced with the addition of the constant exponent. This addition may add the *unbiased* constant exponent to the *biased* input exponent (i.e., the exponent field unprocessed), so the biased exponent result is directly obtained (no need to subtract the bias E_0 as in Fig. 11.7). Then the normalization and rounding hardware of Fig. 11.7 can be used. Exceptional case handling is slightly simpler: If the constant C is strictly smaller than 1, no overflow may ever occur in the computation of the product. Likewise, if $C > 1$, underflow (flush to zero) cannot happen.

The remainder of this section elaborates on this solution with further architectural optimization tricks, in particular for the various types of constants studied in Chap. 12.

15.1.1 Floating-Point Multiplication by a Power of Two

If $F_C = 0$, the constant is a power of two and the multiplication resumes to an addition to the exponent field and exceptional case management. It is important to manage this case properly in an HLS context, as multiplications by 2.0 or 0.5 are quite common in actual code. The architecture is given in Fig. 15.2. The tentative biased exponent of the result, $E_R + E_0$, is $w_E + 1$ -bit wide: The extra bit is either a sign bit for underflow detection when $E_C < 0$, or an overflow bit when $E_C > 0$ (we assume that in the case $E_C = 0$, which corresponds to $C = 1$, the compiler has removed the multiplication altogether).

In Fig. 15.2 we denote $E_R + E_0$ as an sfixed number: With this interpretation both cases are detected when $E_R + E_0$ becomes negative. This extra sign bit is exploited by the `exception handling & pack` block to implement Table 11.2, p. 343.

¹ Strictly speaking, as the exponent is constant, the point does not actually float here.

Note that this simple architecture does not manage subnormal properly:
See Sect. 15.1.6 for subnormal handling.

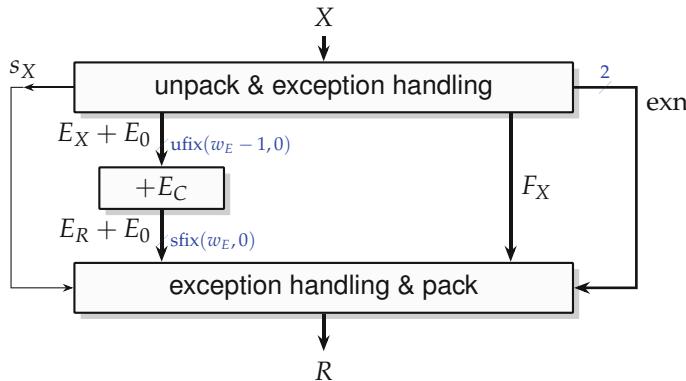


Fig. 15.2 Floating-point multiplier by a power of two.

15.1.2 Baseline Faithful Floating-Point Multiplier by a Constant

This baseline multiplier for the general case $1 < 1.F_C < 2$ is described in Fig. 15.3. It only works with normal numbers:

$$X = (-1)^{s_X} \cdot 2^{E_X} \cdot (1 + F_X) \quad (15.2)$$

$$R = (-1)^{s_R} \cdot 2^{E_R} \cdot (1 + F_R) \quad (15.3)$$

A faithful multiplier may return one of the two floating-point numbers surrounding the exact product. As $1 < 1.F_C < 2$ (strictly) and $1 \leq 1.F_X \leq 2 - 2^{-w_F}$, the exact product is lower-bounded by the floating-point number 1 and upper-bounded by the floating-point number $2 \cdot (2 - 2^{-w_F})$: It requires at most one bit of normalization.

There is one optimization in Fig. 15.3 over Fig. 11.7: It is possible to predict earlier (from the input fraction) if the product P will be normalized or not. For this we define the real number

$$1 + F_{C-1} = \frac{2}{1 + F_C} \in (1, 2) \quad (15.4)$$

(F_{C-1} may also have an infinite binary representation).

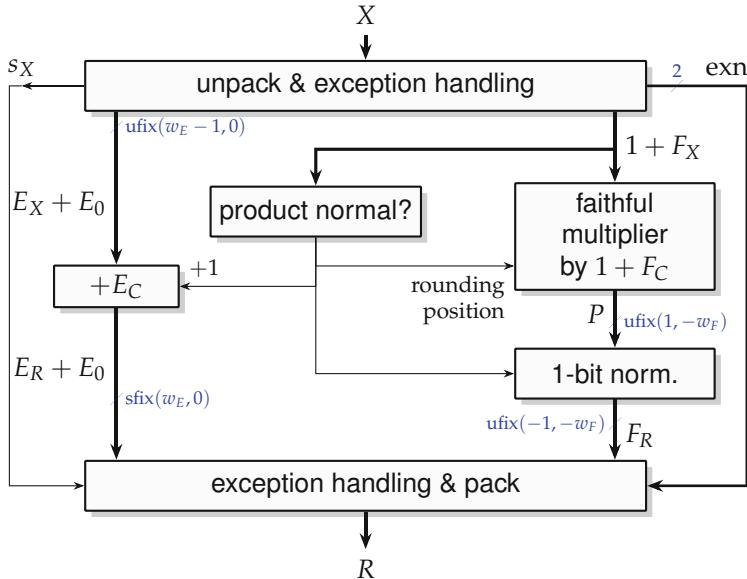


Fig. 15.3 Faithful floating-point multiplier by a real constant.

The **product normal?** block of Fig. 15.3 compares the input fraction F_X to the fraction F_{C-1} (actually to $\lfloor F_{C-1} \rfloor_{-w_F}$, which is F_{C-1} truncated to the format of F_X):

- If $F_X > \lfloor F_{C-1} \rfloor_{-w_F}$, then $F_X \geq \lfloor F_{C-1} \rfloor_{-w_F} + 2^{-w_F} > F_{C-1}$; therefore, $(1 + F_C) \times (1 + F_X) > 2$: The lower of the two possible values that the faithful significand multiplier may return is at least 2, and the product must be shifted right one bit.
- If $F_X < \lfloor F_{C-1} \rfloor_{-w_F}$, which we rewrite $F_X \leq \lfloor F_{C-1} \rfloor_{-w_F} - 2_F^w$, then

$$\begin{aligned}
 (1 + F_C) \times (1 + F_X) &\leq (1 + F_C) \times (1 + \lfloor F_{C-1} \rfloor_{-w_F} - 2_F^w) \\
 &\leq (1 + F_C) \times (1 + F_{C-1} - 2_F^w) \\
 &\leq 2 - 2_F^w (1 + F_C) \\
 &\leq 2 - 2_F^w
 \end{aligned}$$

In this case the product will not need normalization.

There is a limit case $F_X = \lfloor F_{C-1} \rfloor_{-w_F}$ where the product may round up to 2, or round down to $2 - 2_F^w$. This depends on the constant and also on the construction of the faithful multiplier. However, once the constant multiplier is built, it always takes the same rounding decision for this unique value of the fraction. Therefore, it is possible, after the construction of the multiplier,

to acknowledge this decision and to build the [product normal?] block accordingly:

- If, on the input $(1 + \lfloor F_{C-1} \rfloor_{-w_F})$, the multiplier returns 2, then the [product normal?] must compute the test $F_X \geq \lfloor F_{C-1} \rfloor_{-w_F}$.
- If the multiplier returns $2 - 2^w_F$, then the [product normal?] must compute the test $F_X > \lfloor F_{C-1} \rfloor_{-w_F}$.

The result of this test is used to compute the exponent update in parallel to the multiplication, which reduces the critical path.

It is also used to select the position of the rounding bit inside the multiplier itself: position $-w_F$ when the comparison returns false and position $-w_F + 1$ when it returns true. This also reduces the critical path by integrating the rounding addition (performed post-normalization in a standard floating-point multiplier) inside the significand multiplier adder tree or bit heap compression. Area-wise, a post-normalization adder would be at least as large as the constant comparator in the [product normal?] block (see Sect. 6.3.3), so this improvement in delay comes at no cost in area.

In Fig. 15.3, again the tentative biased exponent of the result, E_R is $w_E + 1$ -bit wide: The extra bit is either a sign bit for underflow detection when $E_C < 0$ or an overflow bit when $E_C \geq 0$.

This architecture works as well for IEEEfloat (with subnormals flushed to zero) and with Nfloat, the differences being trivial and located inside the pack and unpack blocks.

15.1.3 Correctly Rounded Multiplication by a Floating-Point Constant

We now address correct rounding, first in the case when the constant C is such that its unique floating-point representation (15.1) has a finite-precision fraction $F_C \in \text{ufix}(-1, -k)$ for some positive integer k .

The main practical application is to multiply by a floating-point constant that fits in the same format as the input. This is useful in an HLS context, where a compiler may attempt to replace a floating-point multiplier with a (cheaper) constant. This is a valid optimization only if the result is bit-for-bit identical to that computed by a software run. There, a floating-point constant (usually in single precision or double precision) is input to a correctly rounded floating-point multiplier. The HLS compiler should therefore build a correctly rounded multiplier by the exact same constant. In this case $k \leq w_F$.

Another application will come in the next section, which reduces a real constant to a finite-precision one (usually with $k > w_F$).

The architecture of a correct multiplier by C is essentially that of Fig. 15.3, with the single difference that an exact constant multiplier is used instead of a faithful one. There is also a small methodological difference in the construction of the [product normal?] block: The behavior on the limit case $F_X = \lfloor F_{C-1} \rfloor - w_F$ can be determined out of the constant only (no need to consider the multiplier construction).

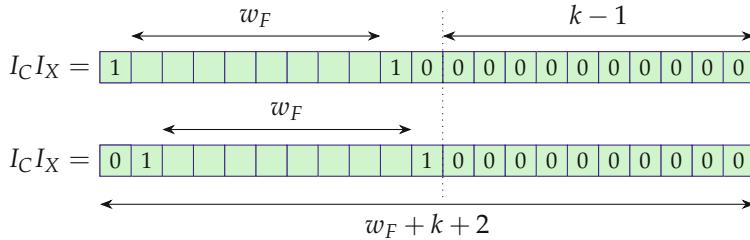


Fig. 15.4 Tie situations in a floating-point multiplier by a floating-point constant.

The exact multiplier internally computes a product on $2 + w_F + k$ bits and rounds it by the addition of a rounding bit, whose position can still be predicted: Again the rounding addition can be merged in the multiplier itself.

This is trivial with round to nearest with ties to away.

For round to nearest with ties to even, before adding sticky bit computation to our architecture, let us first discuss if ties can happen at all. For this it is easier to reason in terms of integer significands: Define $I_X = 2_F^w(1 + F_X)$ and $I_C = 2^k(1 + F_C)$. I_X is an integer in $[2_F^w, 2^{w_F+1} - 1]$. Assuming that k is minimal, I_C is an odd integer in $[2^k, 2^{k+1} - 1]$. The integer significand product $I_C I_X$ fits on $w_F + k + 2$ bits. As Fig. 15.4 shows, tie detection depends on the normalization of the result, but in both the cases a tie has $k - 1$ zeroes at its least significant bit (LSB): The integer product $I_C I_X$ must be a multiple of 2^{k-1} for a tie to happen. Since I_C is odd, this can only happen if I_X is at least 2^{k-1} .

Since $I_X < 2^{w_F+1}$ we immediately deduce that if $k - 1 > w_F$, ties cannot happen. This will be useful in the next section.

If $k - 1 = w_F$, the only value of I_X that can lead to a tie is $I_X = 2_F^w$, which corresponds to a significand equal to 1. In the architecture of Fig. 15.3, it means that the tie situation can be anticipated by precomputing a sticky out of F_X , in parallel to the multiplication. This may slightly reduce the critical path, but not dramatically.

In the remaining cases, $k \leq w_F$, there are fewer cases without tie. Figure 15.4 shows the situations that will lead to a tie: defining $I_{C-1} = 2^k(1 + F_{C-1})$,

- Any $I_X < I_{C-1}$ that is an odd multiple of 2^{k-1} will lead to a tie.
- Any $I_X \geq I_{C-1}$ that is an odd multiple of 2^k will lead to a tie.

If $k = w_F$, we still have a situation without tie for all the constants such that $I_C > \frac{2}{3} \cdot 2^{w_F}$. Indeed, for these constants, $I_{C-1} < \frac{3}{2} \cdot 2^{w_F} = 3 \cdot 2^{w_F-1}$, and there is no odd multiple of 2^{k-1} smaller than $3 \cdot 2^{w_F-1}$.

If $k = w_F$ and $C < \frac{2}{3} \cdot 2^{w_F}$, we have a single tie situation when I_X is $3 \cdot 2_F^w$. Again, this situation can be anticipated by a pseudo-sticky computation on the input of the multiplier. Tie anticipation is still possible when $k < w_F$, but there are more and more values of I_X to filter, hence a diminishing return.

When ties can happen and cannot be anticipated, a standard sticky computation must be implemented on the lower part of the exact product $(1 + F_C) \times (1 + F_X)$, as described in Sect. 11.2.2.

All the previous may seem complicated, but these are simple compile-time computations that enable to optimize the specialized operators.

15.1.4 Correctly Rounded Multiplication by a Real Constant

The construction of a correctly rounded multiplication by an irrational constant C such as π or $\log 2$ has been studied in [BDM08]. The core idea is to determine the smallest value of k such that for all the possible values of $1.F_X$, the correctly rounded product of $1.F_X$ by $\lfloor 1.F_C \rfloor_{-k}$ is the correctly rounded value $1.F_X \times 1.F_C$. Then we can use the architecture of the previous section with $\tilde{C}_k = \lfloor 1 + F_C \rfloor_{-k}$.

It is relatively easy to convince oneself that such a k exists, as follows. As the exponent is irrelevant in this discussion, we assume without loss of generality that $C = 1 + F_C$ and $X = 1 + F_X$. We define *midpoints* as the points equidistant to two consecutive floating-point values (Fig. 15.5). For each value of X we define $\delta(X)$ as the difference between the exact (real) product CX and the nearest midpoint. Since there is a finite number of values of X , the set of all the possible nonzero values of $\delta(X)$ has a minimum element $\bar{\delta}$. No product CX can come closer to a midpoint than $\bar{\delta}$, unless it is exactly a midpoint.

Remark that if $\delta(X)$ were uniformly distributed in $[0, 2^{-w_F-1}]$, we would expect $\bar{\delta}$ to be roughly $1/2_F^w$ of the size of this interval, or $\bar{\delta} \approx 2^{-2w_F-1}$. This statistical assumption is unfounded, but true in practice for irrational constants.

Now, if we are able to approximate the exact product $P = CX$ as some \tilde{P} such that $|\tilde{P} - P| < \bar{\delta}$, then rounding \tilde{P} to the nearest float(w_E, w_F) is equivalent to rounding P , since \tilde{P} and P are on the same side of the nearest midpoint (see Fig. 15.5). Unless \tilde{P} is a midpoint, then P is also necessarily a midpoint.

It is indeed possible to approximate P with arbitrary accuracy: For any integer $k > 0$, we define $\tilde{C}_k = \lfloor C \rfloor_{-k}$, and we define $\tilde{P}_k = \tilde{C}_k \times X$. Then the error $\tilde{P}_k - P$ can be bounded as

$$\begin{aligned} |\tilde{P}_k - P| &= |[\lfloor C \rfloor_{-k}]X - CX| \\ &= |[\lfloor C \rfloor_{-k}] - C| \times (1 + F_X) \\ &< 2^{-k-1} \times 2. \end{aligned}$$

Therefore, choosing $k = \lceil -\log_2 \bar{\delta} \rceil$ ensures $|\tilde{P}_k - P| < \bar{\delta}$; hence, an architecture that rounds \tilde{P}_k to the nearest (using previous section) also rounds P to the nearest. As stated in Sect. 12.3.2, if a shift-and-add graph is used, it may be worth to explore constants with slightly more bits than $\lfloor C \rfloor_{-k}$ as long as they ensure that $|\tilde{P} - P| < \bar{\delta}$.

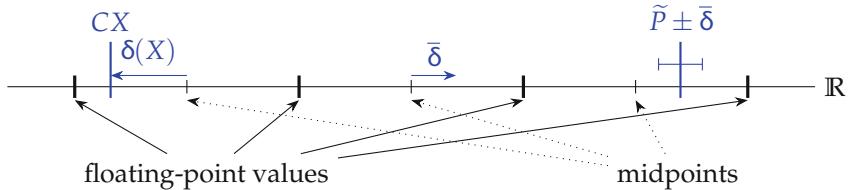


Fig. 15.5 Representation on the real axis of the floating-point numbers, their midpoints, and the worst-case accuracy required for correct rounding.

It remains to determine $\bar{\delta}$, given a constant C . For small values of w_F (up to 24 bits), exhaustive enumeration of all the possible values of F_X is possible, each product CX being evaluated in very high precision (e.g., 512 bits) using fast multiple-precision software. The large precision used adds a margin of 2^{-512} to $\bar{\delta}$, which is not a problem in practice. For larger precisions, a more sophisticated but faster technique using continued fractions can directly compute k [BM08].

From the previous statistical argument, the size of \tilde{C}_k for a correctly rounded multiplication by an irrational C is typically close to $k \approx 2w_F + 1$. As the cost of a constant multiplication is sublinear in the constant size [DIZ07] (see Chap. 12), the price of correct rounding (compared to the naive approach that first rounds to constant to $k = 1 + w_F$ bits and eventually provides a faithful rounding of the product by C) should actually be less than this factor 2 in area – this remains a large price to pay for one more bit of accuracy. The delay overhead will be much smaller, as delay grows logarithmically with precision in all the techniques studied in Chap. 12.

15.1.5 Correct Rounding of the Floating-Point Multiplication by a Rational Constant

Section 12.4 presented a method for multiplying an integer or fixed-point number by a rational A/B that results in efficient architectures when A and B are small. It turns out that in this case, obtaining a correctly rounded result is also fairly cheap thanks to the following theorem (which is a generalization of the “exclusion lemma” used to prove that some division algorithms are correctly rounded [Mul+18]).

Theorem 15.1 *Let X and R be floating-point numbers with w_F fractional bits. If C is a constant obtained by truncating the binary representation of A/B to at least $w_F + 1 + \lceil \log_2 B \rceil$ bits, then rounding the product CX to the nearest floating-point number of precision w_F is equivalent to rounding the exact product $\frac{A}{B}X$.*

Proof By straightforward multiplications by powers of two, we may assume without loss of generality that $X \in [1/2, 1)$ and that both A and B are odd.

Let us use the integral significand representation of the input X , i.e., $X = \frac{I}{2^{w_F}}$ where I is an integer. We want to show that $\frac{A}{B}X$ cannot be too close to the midpoint between two floating-point numbers in the result format. Such a midpoint M can be written $M = \frac{2J + 1}{2^{w_F + 1}}$ for some integer J .

The distance between $\frac{A}{B}X$ and M is therefore written:

$$\begin{aligned} \left| \frac{A}{B}X - M \right| &= \left| \frac{A}{B} \frac{I}{2^{w_F}} - \frac{2J + 1}{2^{w_F + 1}} \right| \\ &= \left| \frac{2AI - (2J + 1)B}{2^{w_F + 1}B} \right| \end{aligned}$$

Here, $2AI$ is an even integer. On the other hand, $(2J + 1)B$ is the product of two odd integers, hence an odd integer. We deduce that their difference is at least one, hence

$$\left| \frac{A}{B}X - M \right| \geq \frac{1}{2^{w_F + 1}B}$$

This defines an “exclusion zone” around midpoints. If we compute $R \approx \frac{A}{B}X$ such that $|R - \frac{A}{B}X| < \frac{1}{2^{w_F + 1}B}$, rounding R to w_F bits is then equivalent to rounding $\frac{A}{B}X$. Truncating the infinite representation of $\frac{A}{B}$ to the precision $2^{-w_F - 1 - \lceil \log_2 B \rceil}$ provides this accuracy.

Remark that a floating-point fraction is a particular case of rational constants, with a power of two at the denominator. Theorem 15.1 is mostly use-

less in this case: All it tells us is that the significand product should be performed exactly.

15.1.6 Subnormal Handling

Subnormals, as in the case of standard floating-point multipliers, require additional shifters. The optimization opportunity is that the range of these shifts is limited by the exponent of the constant.

Here we must distinguish the cases $C > 1$ and $C < 1$.

15.1.6.1 Case $C > 1$

If $C > 1$, the multiplication by C will never produce a subnormal out of a normal input. However, input subnormals must be handled, as some may become normal outputs. More precisely, let us define n_X the “is normal” bit of the input and λ the leading zero count on the input significand $n_X + F_X$. For normal inputs we have $\lambda = 0$; for subnormal inputs $\lambda > 0$. We can distinguish the following cases:

- When $\lambda > E_C$, the multiplication by $C = 2^{E_C}(1 + F_C)$ is performed by first multiplying $0 + F_X$ by 2^{E_C} (this is a shift left of $0 + F_X$ by E_C bits, and $E_C \geq 0$ since $C > 1$). This shift will not yield a normal significand, so we may simply input $0 + 2^{E_C}F_X$ to the significand multiplier that will complete the multiplication $(0 + 2^{E_C}F_X) \times (1 + F_C)$. By construction the product will be smaller than 2 in this case: It does not require post-normalization. However, the product may become normal when $\lambda = E_C + 1$: In this case the exponent field must be incremented.
- When $0 \leq \lambda \leq E_C$, we know that output will be a normal number. To handle this case, a normalizer (leading counter and shifter) builds a normal version of $0 + F_X$ as $F_X = 2^{-\lambda}(1 + F'_X)$, which is a normal floating-point number that can be input to the architecture of previous sections. The exponent adder must add $E_C - \lambda$ to the biased exponent $E_X + E_0$.

As Fig. 15.6 shows, both cases can be handled by a single normalizer that shifts left the input $n_X + F_X$ by at most E_C bits and also outputs the shift amount $\lambda' = \max(\lambda, E_C)$ to be accounted for in the exponent processing. This normalizer does nothing ($\lambda' = 0$) for normal inputs. For subnormal inputs such that the product remains subnormal, it saturates to $\lambda' = E_C$, so the eventual exponent field is unchanged, except in the case when a subnormal input yields a normal output, in which case the exponent field is incremented. The significand output of this normalizer is simply fed to the architecture of previous sections.

This normalizer adds to the critical path, but for many practical constants, E_C will be much smaller than w_F , the maximum shift of the corresponding normalizer in a standard floating-point multiplier.

15.1.6.2 Case $C < 1$

If $C < 1$ (therefore $E_C < 0$), the multiplication by C will never normalize a subnormal number, but it may produce a subnormal out of a normal input. This case is managed by post-processing the tentative biased exponent E_R and fraction the exact product P : If $E_R < e_{\min}$, then the product must be shifted right by $\lambda = e_{\min} - E_R$. Unfortunately, it is useless in this case to perform the shift before the multiplier, since we are not allowed to discard any shifted bit.

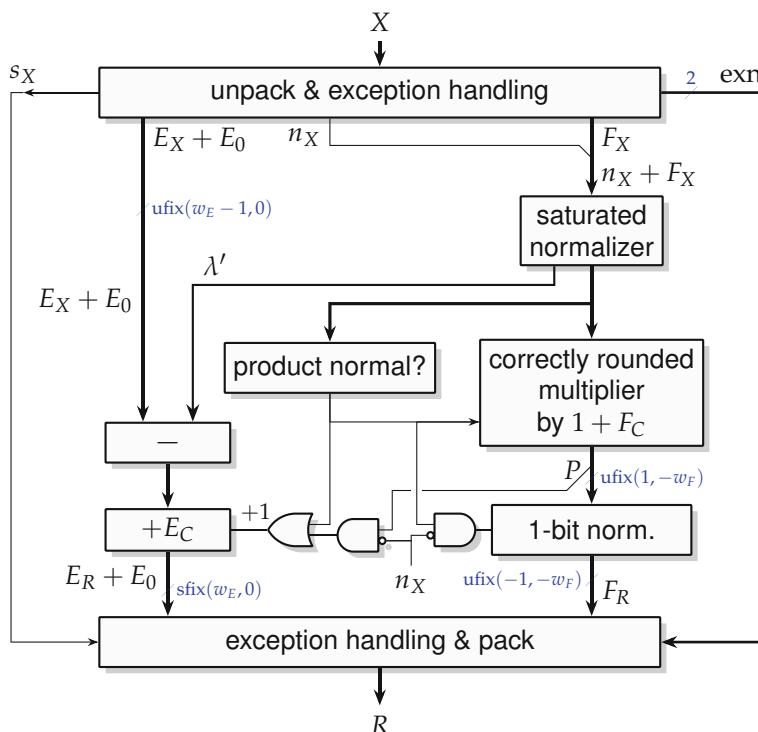


Fig. 15.6 Correctly rounded constant floating-point multiplier with subnormal management in the case $C > 1$.

In this case we must revert to an architecture that performs rounding after normalization. Then the same shifter may be used for input subnormals, always shifting right by $-E_C$ bits.

We do not detail the architecture here. It may still benefit from the optimization of the constant significand multiplier and from the limited shift distance when the constant exponent is smaller in magnitude than the significand size w_F .

15.2 Floating-Point Division by a Small Integer Constant

This section shows how to build a divider of a floating-point input X by a small integer constant D that ensures bit-for-bit compatibility with an IEEE 754-compliant divider. For this purpose, it builds upon the Euclidean dividers by a small constant of Chap. 13. A compiler of floating-point code to FPGA can use this operator as a drop-in replacement for a generic divider, at a fraction of the cost.

For simplicity we write this section for an odd integer constant D . We will then show how it can be trivially extended to a multiplication with any $2^k D$ where k is an integer and D is an odd integer. For instance, the architectures presented here work equally well for dividing by 12 or 0.75.

15.2.1 Baseline Operator for Normalized Inputs

A normal floating-point input X is given by its fraction and exponent:

$$X = 2^{E_X} (1 + F_X) \quad \text{with } 1 + F_X \in [1, 2) . \quad (15.5)$$

As D is an odd integer, its floating-point representation is:

$$D = 2^{E_D} (1 + F_D) \quad \text{with } 1 + F_D \in [1, 2) . \quad (15.6)$$

In other words, D is an integer of $E_D + 1$ bits, or F_D can be represented as E_D -bit fraction. For instance, $5 = 2^2(1.01_2)$: $F_D = .01_2$ fits on $E_D = 2$ bits.

The main issues to address are the normalization and rounding of the floating-point division of X by D .

15.2.1.1 Normalization

Let us write the division

$$\frac{X}{D} = \frac{2^{E_X} (1 + F_X)}{2^{E_D} (1 + F_D)} = 2^{E_X - E_D} \cdot \frac{1 + F_X}{1 + F_D} . \quad (15.7)$$

As $\frac{1+F_X}{1+F_D} \in [0.5, 2)$, this is almost the normalized mantissa of the floating-point representation of the result:

- If $F_X \geq F_D$, then $\frac{1+F_X}{1+F_D} \in [1, 2)$; the mantissa is correctly normalized and the floating-point number to be returned is

$$Y = \left\lfloor \frac{2^{E_D}(1+F_X)}{D} \right\rfloor_{-w_F} 2^{E_X-E_D} . \quad (15.8)$$

- If $F_X < F_D$, then $\frac{1+F_X}{1+F_D} \in [0.5, 1)$, and to normalize it the best option is to left-shift the input significand by one bit. Thus, the floating-point number to be returned is

$$Y = \left\lfloor \frac{2^{E_D+1}(1+F_X)}{D} \right\rfloor_{-w_F} 2^{E_X-E_D-1} . \quad (15.9)$$

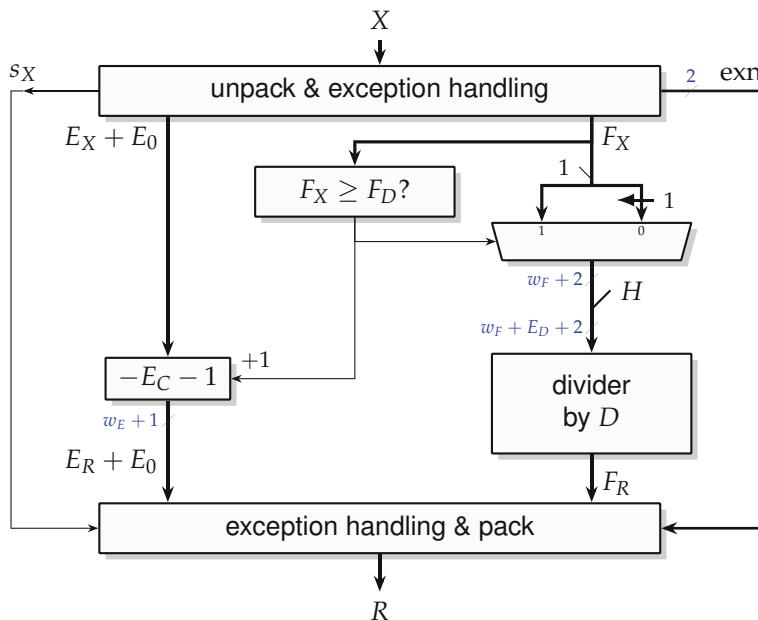


Fig. 15.7 Floating-point division by a small constant.

The comparison between F_X and F_D is extremely cheap as long as D is a small integer, because F_D fits on E_D nonzero bits. Thus, the comparison is reduced to the comparison of these E_D bits to the leading E_D bits of F_X .

15.2.1.2 Rounding

Let us now address the issue of correctly rounding the mantissa fraction. If we ignore the remainder, the obtained result is the rounding toward zero of the floating-point division.

To obtain correct rounding to the nearest, a first idea is to consider the final remainder. If it is larger than $D/2$, we should round up, i.e., increment the mantissa. The comparison to $D/2$ could cost nothing: The rightmost table of Fig. 13.3, p. 433, could hold the result of this comparison instead of the remainder value. However, it would be followed by a significand-wide addition.

A better idea is to use the identity $\lfloor z \rfloor = \lfloor z + \frac{1}{2} \rfloor$, which in our case becomes

$$\left\lfloor \frac{2^{E_D+\epsilon}(1+F_X)}{D} \right\rfloor = \left\lfloor \frac{2^{E_D+\epsilon}(1+F_X) + D/2}{D} \right\rfloor \quad (15.10)$$

with ϵ being 0 if $F_X \geq F_D$ and 1 otherwise. Furthermore, in the floating-point context we may assume that D is odd, since powers of two are managed as exponents: Let us write $D = 2H + 1$. We obtain

$$\left\lfloor \frac{2^{E_D+\epsilon}(1+F_X)}{D} \right\rfloor = \left\lfloor \frac{2^{E_D+\epsilon}(1+F_X) + H}{D} + \frac{1}{2D} \right\rfloor \quad (15.11)$$

$$= \left\lfloor \frac{2^{E_D+\epsilon}(1+F_X) + H}{D} \right\rfloor \quad (15.12)$$

so instead of adding a round bit to the result, we may add H to the dividend before its input into the integer divisor. It seems we have not won much, but this pre-addition is actually for free: The addend $H = (D-1)/2$ is an E_D -bit number, and we have to add it to the significand of X that is shifted left by $E_D + \epsilon$ bits – it is a mere concatenation at the LSB. Thus, it is possible to save both the area and the latency of the rounding adder.

To sum up, the management of a floating-point input adds to the mantissa divider one (small) exponent adder and one (large) mantissa multiplexer, as illustrated by Fig. 15.7. In this figure, exn is a 2-bit exception vector used to represent 0, $\pm\infty$, and NaN (not a number). As D is an odd integer, tie situations cannot happen if the input and output formats are identical.

15.2.1.3 Dividing by a Scaled Odd Integer

To divide by $D' = 2^k D$ where D is an odd integer, the only change is that the exponent adder adds $k - E_C - 1$ instead of $-E_C - 1$. If $k \geq E_C + 1$, the operator may overflow, but then it can no longer underflow: In this case the extra bit computed by the exponent adder is interpreted as overflow by the exception handling logic.

Hands on: Floating-point division by a small constant in FloPoCo

The following command builds a correctly rounded floating-point divider by 3, as often needed by 1-D stencil computations:

```
flopoco FPConstDiv we=8 wf=23 d=3
```

To divide by 10, the current interface is disputable: The user has to provide the constant as an odd integer with an exponent (here $10 = 5 * 2^1$):

```
flopoco FPConstDiv we=8 wf=23 d=5 dExp=1
```

A console message will confirm that the operator is dividing by 10.

To divide by 9 (as needed by some 2D stencil computations), it is beneficial to use a composition of dividers by 3, as discussed in Sect. 13.5. The interface for this is identical to that of IntConstDiv:

```
flopoco FPConstDiv we=8 wf=23 d=3 : 3
```

15.2.2 Subnormal Handling

Division by a constant is, from this point of view, perfectly equivalent to a multiplication by the inverse: All the discussion of Sect. 15.1 applies.

15.3 Floating-Point Squares, Cubes, and X^p

An X^p operator is essentially useful for small values of p , and this is what we address in this section. The management of floating point for these operations is straightforward. From

$$X = (-1)^{s_X} \cdot 2^{E_X} \cdot (1 + F_X) \quad (15.13)$$

it comes

$$X^2 = 2^{2E_X} \cdot (1 + F_X)^2 \quad (15.14)$$

$$X^3 = (-1)^{s_X} \cdot 2^{3E_X} \cdot (1 + F_X)^3 \quad (15.15)$$

$$X^p = (-1)^{ps_X} \cdot 2^{pE_X} \cdot (1 + F_X)^p \quad . \quad (15.16)$$

A floating-point squarer is really a specialization of the multiplier. As $(1 + F_X)^2 \in [1, 4]$, a 1-bit normalization is needed (one multiplexer), and overflow/underflow processing is similar to what it is in a multiplier.

In the general case X^p for $p > 0$, the sign of the result is positive if p is even and equal to the sign of X if p is odd. Exponent processing is sim-

ilarly simple, requiring a multiplier by a small constant, which resumes to very few shifts and additions (see Chap. 12). Overflow and underflow processing is straightforward. Note that the processing of input subnormals (if any) is trivial since on such an input X^p rounds to zero for any $p \geq 2$. However, if subnormals are to be supported in output, they will require specific hardware for saturating the exponent to e_{\min} and denormalizing the output significand.

Let us now address the significand processing, which consists in computing $(1 + F_X)^p$. As $(1 + F_X)^p \in [1, 2^p]$, a p -bit normalization is needed. It may use a normalizer from Sect. 10.3, whose leading zero count output is used to update the exponent. If latency is important, it is possible to predict the leading bit of the output significand by a sequence of comparisons. For instance:

- For X^2 we need a 1-bit normalization if $F_X > \sqrt{2} - 1$.
- For X^3 we need a 1-bit normalization if $\sqrt[3]{2} - 1 < F_X < \sqrt[3]{4} - 1$ and a 2-bit normalization if $F_X > \sqrt[3]{4} - 1$.

In the general case we need comparisons of F_X to $\sqrt[p]{2^i} - 1$ for $i \in \{1, \dots, p-1\}$ (these exponent borders are shown in blue in Fig. 15.8). All these comparisons can be performed in parallel and their result priority-encoded, all in parallel with the significand processing. This way the exponent processing is not on the critical path. All this is a generalization of the [product normal?] blocks in previous sections.

The exact value of $(1 + F_X)^p$ is a number of $p \times (1 + w_F)$ bits. It will therefore be costly to compute, although it is much cheaper than $p - 1$ generic multipliers thanks to the square-and-multiply technique of Sect. 14.3.2: for instance M^5 is computed as $(M^2)^2 \times M$ using two squarers (of increasing sizes) and one multiplier.

For a faithful result, the function $f(x) = (1 + x)^p$ for $x \in [0, 1]$ is well suited to some of the methods studied in Part III of this book, in particular polynomial approximation [PBM01]. Indeed, the plots of Fig. 15.8 show that these functions do not exhibit any singularity. Generic function approximation methods are in principle cheaper than the exact computation of $(1 + F_X)^p$, although the latter is required to achieve correct rounding.

15.4 Floating-Point Addition of Positive Terms

The application-specific context may dictate that the values to be added are all of the same sign. It is, for instance, often the case when probabilities are involved, either to be added, or to be multiplied in the log domain. In this case there can be no cancellation, which means that the [LZC+shifter] blocks visible in Fig. 11.5, p. 337, can be removed altogether. Another point of view

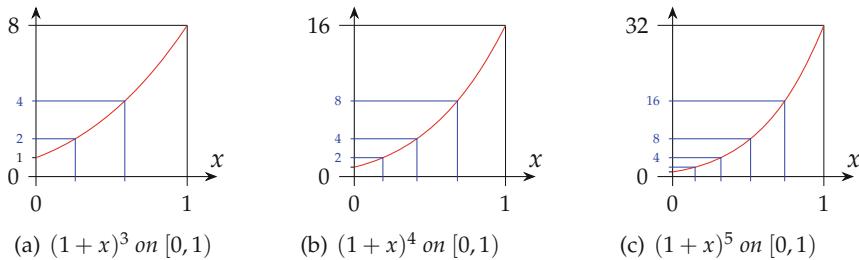


Fig. 15.8 The functions to evaluate in the significand path of an X^p operator.

is that the adder only needs the far path of Fig. 11.6, p. 341. This saves about 1/3 the hardware of a generic floating-point adder [Som+20].

The same idea is exploited in fused architecture for the sum of squares [DKP09; MS13] reviewed below in Sect. 15.6.

15.5 Combined Floating-Point Sum and Difference

If both $X - Y$ and $X + Y$ have to be computed (as is the case in an implementation of the fast Fourier transform (FFT) for instance), several optimizations can be implemented [TDX12]:

- The operand decoding and the initial shift in Fig. 11.5, p. 337, are shared.
- At most one of the operations may be an effective subtraction and thus require the close path. However, it is possible that both results require a far path.

Figure 11.6, p. 341 therefore becomes a 3-path architecture comprising a *close* path and two *far* paths that share an alignment shifter [TDX12].

15.6 Fused Multiply and Add, Other Sums of Products

The Fused Multiply-Add (FMA) computes $A \times B + C$ with a single rounding of the exact result. We mention it here for completeness and because it has become part of the IEEE 754 standard, but it is not an application-specific operator, quite the contrary: the FMA has been introduced as a universal operator that replaces addition ($A \times 1 + C$) and multiplication ($A \times B + 0$) and allows for flexible implementations of division and square root [Mar00; CHT02; Mul+18]. However, an FMA is more expensive than an adder and a multiplier combined, as the requirement of a single rounding mandates a wider internal datapath. Many FMA implementations have been described [MHR90; HMC90; Sei03; LB04; BL05; SST05; Yu+06; Tro+07; QSL07; Lut11]

and we refer the interested reader to these works, or to textbooks [Par10; Mul+18].

Other fusions and specializations of sum of products have been studied, in particular the fused sum of two products [SS08; Tao+13], the sum of squares [DKP09; MS13; Tao+13], and the sum of three [Tao+12] or four [Tao+13] floating-point numbers.

Some of these works implement correctly rounded fused operators. Faithful rounding may allow for significantly cheaper operators, but only for a sum of numbers of the same sign (including sum of squares [DKP09]). Otherwise, there is a possibility of large cancellation, and managing it properly requires an exact computation of most of the bits of the exact result. Therefore, there is little to save by aiming for a faithfully rounded operator.

Conversely, a faithful floating-point sum of three squares was used as an example of operator fusion in the introduction of this book; see Fig. 1.5, p. 12. It does save both area and latency compared to a combination of adders and multipliers.

15.7 Floating-Point Optimizations in an HLS Context

In high-level synthesis (HLS), a compiler is in charge of turning a C or C++ program into hardware. This may involve many optimizations, including all the operator specializations we have seen so far. Such optimizations may be classified into two classes: those which strictly preserve the semantics of the C program and those which do not, which are often called “unsafe” optimizations.² For instance, in the GNU compiler collection (GCC) suite (and other mainstream compilers that inherit some of its options), the `-funsafe-math-optimizations` compilation flag allows the compiler to relax conformance to the language standard. This includes reparenthesizing expressions as if floating-point additions and multiplications were associative, replacing a signed zero by the other, and degrading the accuracy of some operations or functions (e.g., using faithful rounding instead of correct rounding).

At the time of writing this book, mainstream HLS compilers are far from performing all the arithmetic optimizations that would be possible for hardware, in part because of their software heritage [Ugu+20].

To conclude this chapter, let us consider some “safe” floating-point optimizations that are specific for hardware. Most programming languages, including C and C++, base their floating-point semantics on the IEEE 754 standard. Therefore, specialized adders, multipliers, dividers, or comparators must implement IEEE semantics to be considered “safe.” More inter-

² Of course, trusting the compiler to respect language standards is necessary but not sufficient to ensure safe applications.

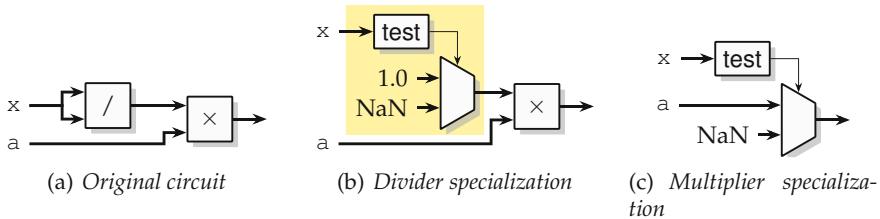


Fig. 15.9 Optimization opportunities for floating-point $x / x * a$.

estingly, some floating-point optimizations are not allowed in “safe” mode. Here are some examples that can be found in the C standard [StdC18]:

- x/x and 1.0 are not equivalent if x is zero, infinite, or NaN (in which case the value of x/x is NaN).
 - $x - y$ and $-(y - x)$ are not equivalent because $1.0 - 1.0$ is $+0$ but $-(1.0 - 1.0)$ is -0 (in the default rounding direction).
 - $x - x$ and 0 are not equivalent if x is a NaN or infinite.
 - $0 \times x$ and 0 are not equivalent if x is a NaN, infinite, or -0.
 - $x + 0$ and x are not equivalent if x is -0, because $(-0) + (+0)$, in the default rounding mode (to the nearest), yields $+0$, not -0 .
 - $0 - x$ and $-x$ are not equivalent if x is $+0$, because $-(+0)$ yields -0 , but $0 - (+0)$ yields $+0$.

Of course, programmers usually do not write x/x or $x + 0$ in their code. However, such computations will arise as side effects of other optimization steps, such as code hoisting, or procedure specialization and cloning [Muc97]. Their optimization is therefore relevant in the context of a global optimizing compiler.

Let us consider the first example (the others are similar): A compiler is not allowed to replace x/x with 1.0 unless it is able to prove that x will never be zero, infinity, or NaN. This is true for HLS as well as for a standard compiler. However, it could replace x/x with something like `(is_zero(x) || is_infty(x) || is_nan(x) ? NaN : 1.0)`. This is, to our knowledge, not implemented in the current compiler: In software targeting a modern superscalar processor, the test on x may well be more expensive than the division.

However, if implemented in hardware, this test is quite cheap: It consists in detecting if the exponent bits are all zeroes (which capture the 0 case) or all ones (which captures both infinity and NaN cases). The exponent is only 8 bits for single precision and 11 bits for double precision.

In an FPGA context, it therefore makes perfect sense to replace x/x (Fig. 15.9a) with an extremely specialized divider depicted in Fig. 15.9b. Furthermore, the two possible values are interesting to propagate further (1.0 because it is absorbed by multiplication, NaN because it is extremely conta-

gious). Therefore, this optimization step may enable further ones, where the multiplexer is pushed down the computation, as illustrated by Fig. 15.9c.

This example shows that there are many floating-point optimization opportunities that arise when targeting hardware that were not relevant in software.

References

- [BDM08] Nicolas Brisebarre, Florent de Dinechin, and Jean-Michel Muller. “Integer and Floating-Point Constant Multipliers for FPGAs”. In: *International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE, 2008, pp. 239–244. (cit. on p. 459).
- [BL05] Javier D. Bruguera and Tomás Lang. “Floating-point Fused Multiply-Add: Reduced Latency for Floating-Point Addition”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2005. (cit. on p. 469).
- [BM08] Nicolas Brisebarre and Jean-Michel Muller. “Correctly Rounded Multiplication by Arbitrary Precision Constants”. In: *IEEE Transactions on Computers* 57.2 (2008), pp. 165–174. (cit. on p. 460).
- [CHT02] Marius Cornea, John Harrison, and Ping Tak Peter Tang. *Scientific Computing on Itanium®-Based Systems*. Intel Press, 2002. (cit. on p. 469).
- [DIZ07] Vassil Dimitrov, Laurent Imbert, and Andrew Zakaluzny. “Multiplication by a Constant is Sublinear”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2007, pp. 261–268. (cit. on p. 460).
- [DKP09] Florent de Dinechin, Cristian Klein, and Bogdan Pasca. “Generating High-Performance Custom Floating-Point Pipelines”. In: *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2009, pp. 59–64. (cit. on pp. 469, 470).
- [HMC90] E. Hokenek, R. K. Montoye, and P. W. Cook. “Second-Generation RISC Floating Point with Multiply-Add Fused”. In: *IEEE Journal of Solid-State Circuits* 25.5 (1990), pp. 1207–1213. (cit. on p. 469).
- [LB04] Tomás Lang and Javier D. Bruguera. “Floating-Point Multiply-Add-Fused with Reduced Latency”. In: *IEEE Transactions on Computers* 53.8 (2004), pp. 988–1003. (cit. on p. 469).
- [Lut11] David Lutz. “Fused Multiply-Add Microarchitecture Comprising Separate Early-Normalizing Multiply and Add Pipelines”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2011, pp. 123–128. (cit. on p. 469).

- [Mar00] Peter Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000. (cit. on p. 469).
- [MHR90] R. K. Montoye, E. Hokonek, and S. L. Runyan. “Design of the IBM RISC System/6000 floating-point execution unit”. In: *IBM Journal of Research and Development* 34.1 (1990), pp. 59–70. (cit. on p. 469).
- [MS13] Jae Hong Min and Earl E. Swartzlander. “Fused Floating-Point Two-term Sum-of-Squares Unit”. In: *Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE, 2013. (cit. on pp. 469, 470).
- [Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. (cit. on p. 471).
- [Mul+18] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. 2nd ed. Birkhäuser Boston, 2018. (cit. on pp. 461, 469, 470).
- [Par10] Behrooz Parhami. *Computer Arithmetic, Algorithms and Hardware Designs*. 2nd ed. Oxford University Press, 2010. (cit. on p. 470).
- [PBM01] José-Alejandro Piñeiro, Javier D. Bruguera, and Jean-Michel Muller. “Faithful Powering Computation Using Table Look-Up and a Fused Accumulation Tree”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2001, pp. 40–47. (cit. on p. 468).
- [QSL07] E. Quinnell, E. E. Swartzlander, and C. Lemonds. “Floating-Point Fused Multiply-Add Architectures”. In: *Asilomar Conference on Signals, Circuits and Systems*. IEEE, 2007, pp. 331–337. (cit. on p. 469).
- [Sei03] Peter-Michael Seidel. “Multiple Path IEEE Floating-Point Fused Multiply-Add”. In: *46th International Midwest Symposium on Circuits and Systems*. IEEE, 2003, pp. 1359–1362. (cit. on p. 469).
- [Som+20] Lukas Sommer, Lukas Weber, Martin Kumm, and Andreas Koch. “Comparison of Arithmetic Number Formats for Inference in Sum-Product Networks on FPGAs”. In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 75–83. (cit. on p. 469).
- [SS08] Hani H. Saleh and Earl E. Swartzlander. “A Floating-Point Fused Dot-Product Unit”. In: *International Conference on Computer Design (ICCD)*. 2008, pp. 426–431. (cit. on p. 470).
- [SST05] Eric M. Schwarz, Martin Schmookler, and Son Dao Trong. “FPU Implementations with Denormalized Numbers”. In: *IEEE Transactions on Computers* 54.7 (2005), pp. 825–836. (cit. on p. 469).
- [StdC18] ISO/IEC. *International Standard ISO/IEC 9899:2018. Programming languages – C*. 2018. (cit. on p. 471).

- [Tao+12] Yao Tao, Gao Deyuan, Fan Xiaoya, and Ren Xianglong. "Three-Operand Floating-Point Adder". In: *12th International Conference on Computer and Information Technology*. 2012, pp. 192–196. (cit. on p. [470](#)).
- [Tao+13] Yao Tao, Gao Deyuan, Fan Xiaoya, and Jari Nurmi. "Correctly Rounded Architectures for Floating-Point Multi-Operand Addition and Dot-Product Computation". In: *Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE, 2013. (cit. on p. [470](#)).
- [TDX12] Yao Tao, Gao Deyuan, and Fan Xiaoya. "A Multi-Path Fused Add-Subtract Unit for Digital Signal Processing". In: *Computer Science and Automation Engineering (CSAE)*. 2012. (cit. on p. [469](#)).
- [Tro+07] S. D. Trong, Martin M. Schmookler, E. M. Schwarz, and M. Kroener. "P6 Binary Floating-Point Unit". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2007, pp. 77–86. (cit. on p. [469](#)).
- [Ugu+20] Yohann Uguen, Florent de Dinechin, Victor Lezaud, and Steven Derrien. "Application-Specific Arithmetic in High-Level Synthesis Tools". In: *ACM Transactions on Architecture and Code Optimization* 17.1 (2020). (cit. on p. [470](#)).
- [Yu+06] X. Y. Yu, Y.-H. Chan, B. Curran, E. Schwarz, M. Kelly, and B. Fleischer. "A 5GHz+ 128-bit Binary Floating-Point Adder for the POWER6 Processor". In: *European Solid-State Circuits Conference*. 2006, pp. 166–169. (cit. on p. [469](#)).

Part III

**Generic Methods for Fixed-Point
Function Approximation**



16

CHAPTER 16

Generalities on Fixed-Point Function Approximation

With their heads bent down over their electric computers, thirty scientific men were absorbed in transcendental calculations.

Jules Verne, *In The Year 2889*

This chapter presents several classes of numerical (real-valued) functions and introduces issues associated with their implementation in fixed point. These issues are illustrated with the simplest of these implementations: plain tabulation.

Many applications require the evaluation of a mathematical function of one variable. Here are but a few examples:

- Fixed-point sine, cosine, exponential, and logarithms are routinely used in signal processing algorithms.
- Approximations to the reciprocal $1/x$ and reciprocal square root $1/\sqrt{x}$ functions are used in some floating-point units to bootstrap division and square root computation [Mar00].
- Random number generators with a Gaussian distribution may be built using the Box-Muller method, which requires logarithm, square root, sine, and cosine [Lee+06]. Arbitrary distributions may be obtained by the inversion method, in which case one needs a fixed-point evaluator for the inverse cumulative distribution function (ICDF) of the required distribution [Che+07]. There are as many ICDF as there are statistical distributions.
- Floating-point elementary function hardware architectures often reduce to evaluations of the function (or a range-reduced variant) on a fixed-point interval [DD07b; DD07a].

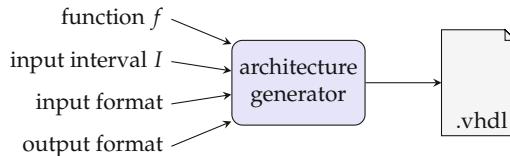


Fig. 16.1 Parameters of a generator of fixed-point function approximators.

- The functions $\log_2(1 + 2^x)$ and $\log_2(1 - 2^x)$ are used to build adders and subtracters in the LNS [VCA10], and many more functions are needed for complex LNS [AC11].
- In general, many scientific applications directly rely on elementary functions that often become the computational bottleneck. For instance, SPICE electronic circuit simulations make extensive use of exponential and logarithm functions [KD09].

This part of the book describes various *generic* function approximation methods. A method is generic if the function itself is an input, as illustrated by Fig. 16.1. A generic method may address all the needs listed above.

There also exist function-specific techniques, applicable only to a given function or class of functions. The best known example is probably CORDIC [Meh+09] for trigonometric functions and its variations for exponential, logarithms, and others. These are well covered in Muller's reference textbook [Mul16]. Which technique applies to which function depends on the class of the function and will be briefly discussed in Sect. 16.3.

The simplest functions are *functions of one variable*, and most techniques presented in this part apply only to functions of one variable: Either these techniques scale poorly to functions of several variables or they simply do not work. For functions of several variables, the common practice is therefore to first attempt to express them in terms of functions of one variable. In the general case, such reductions can be found with the help of mathematical books (the reference being a 1964 book by Abramowitz and Stegun [AS64], which can now be found online,¹) or specialized websites such as MathWorld² or the Mathematical Functions Grimoire.³

Finally, in this part, we only deal with fixed-point input and outputs, for two reasons. First, fixed-point function evaluators are useful in their own right in many applications, in particular in signal processing. Second, fixed-point function evaluators are a core building block when implementing floating-point functions, as will be illustrated in Chap. 22 of this book. For illustration, to build a *floating-point* universal function approximator [Tho15], the algorithm actually starts by decomposing the function's input domain

¹ <http://people.math.sfu.ca/~cbm/aands/>.

² <http://mathworld.wolfram.com/>.

³ <http://fungrim.org/>.

into binades, i.e., intervals where the exponent is constant. These are effectively fixed-point domains, thus reducing the problem to a fixed-point function approximation.

For our readers more interested in floating point, a general introduction to floating-point implementation of numerical functions can be found in Muller's reference textbook [Mul16], in the Handbook of Floating-Point Arithmetic [Mul+18]. Implementation details are covered in more specialized books such as Markstein's [Mar00] or the one by Cornea, Harrison, and Tang [CHT02].

The fixed-point formats of the inputs must match the *input domain* on which the function is evaluated. The format of the output must match the *output range* of the function. We now discuss these issues in more detail.

16.1 Defining Domain and Range

It may seem natural that a generator inputs the specification of the interval $[a, b]$ on which the function is to be evaluated. Indeed, early generators [Lee+05; DT05] would input the interval bounds a and b .

However, the fact that the function f itself is an input has one beneficial side effect: It allows us to integrate, in this function specification, a change of variable that transforms its input domain into a standard one. Indeed, in this chapter and the following, we will assume (almost without loss of generality, this will be discussed below) that the function is defined on the interval $[0, 1]$ or $[-1, 1]$. In other words, the specification of the input interval I in Fig. 16.1 is reduced to a Boolean choice between a signed and an unsigned input. This is illustrated by Fig. 16.2.

These input intervals are chosen because they correspond to fixed-point formats. More precisely, for a precision ℓ (a negative integer) the input interval $[0, 1]$ maps to the ufix($-1, \ell$) format, while $[-1, 1]$ maps to the sfix($0, \ell$) format. The fact that these intervals that are open in 1 reflects that the largest value in each format is not exactly 1, but $1 - 2^\ell$ (see Chap. 2). This subtlety and its management will be discussed in detail in the next section. Let us first focus on how to manage a function when the needed domain is larger or smaller than $[0, 1]$ or $[-1, 1]$.

The main idea here is that the corresponding scaling (or more generally the required change of variable) can simply be integrated in the definition of the function. This is best described on the following example: Consider the well-known sine function, whose graph is depicted in Fig. 16.3.

The input domain of the sine function is \mathbb{R} . If we restrict the sine function to $[-1, 1]$, we obtain Fig. 16.4a. This particular interval is probably of little practical use. In most applications, we will need the sine on one period, e.g., on the interval $[-\pi, \pi]$. This does not match well with a fixed-point format, because π is not a power of 2. The simplest solution in practice will therefore

be to consider $\sin(\pi x)$ on $[-1, 1]$, as illustrated by Fig. 16.4b: There, $[-1, 1]$ represents one period of the transformed function. However, there are a lot of symmetries on this useful interval; therefore, it will in practice be enough to evaluate the sine on a quarter of its period, the values on the three other quarters being reconstructed out of this quarter—Chap. 20 will study this in detail. The function that remains to be evaluated in this case is $\sin(\frac{\pi}{2}x)$ on $[0, 1]$, as illustrated by Fig. 16.4c.

As a function generator inputs a specification of the function, it may as well input $\sin(\frac{\pi}{2}x)$ instead of $\sin(x)$. The experience of the authors is that inputting interval bounds to the function generator only adds redundancy and complexity to the code of the generator.

There is another beneficial effect in standardizing on the $[0, 1]$ and $[-1, 1]$ intervals: These are fixed-point intervals that are asymmetric in a well-defined way. This allows for a clear statement of the subtle issues related to open or closed intervals, which we now review.

16.2 Discretization Issues

We now review the possible issues that may arise when mapping the continuous range and domain of the function to the corresponding discrete fixed-point range and domain. It is important to understand them well: As we will see, ignoring them can have catastrophic consequences, for instance, returning one end of the domain instead of the other end, because of the modulo property of fixed-point arithmetic.

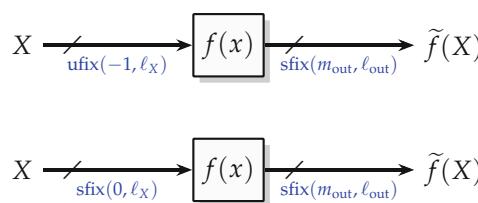


Fig. 16.2 Black-box of a fixed-point function approximator, for unsigned (top) and signed (bottom) input.

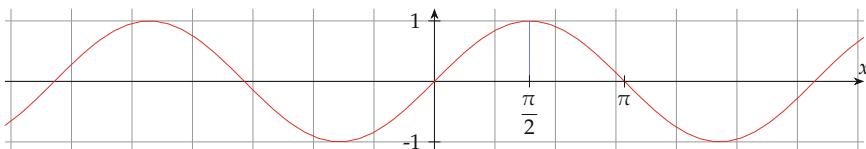


Fig. 16.3 The sine function.

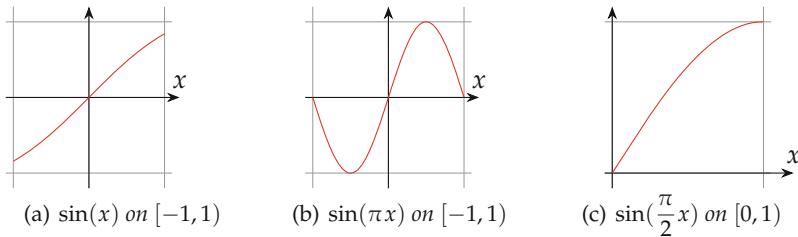


Fig. 16.4 Various domains for the sine function.

In this section, we assume that the reader is familiar with the notions of ulp, rounding to the nearest, and faithful rounding, introduced in Sect. 3.1.

16.2.1 Range Issues and Their Solutions

A first issue a designer must be aware of is that our intervals are open at 1. This is of course a consequence of the choice we made, but the corresponding problems are in fact deeply related to fixed-point representations and would similarly arise for other conventions.

Again, let us take the example of the sine function, restricted to the domain of Fig. 16.4c, but now shown along with the discrete fixed-point grid. Figure 16.5 shows the discretization of $\sin(\frac{\pi}{2}x)$ on $[0, 1)$ when both input and output intervals are mapped on the ufix($-1, -4$) format (see Sect. 2.2.1). In this figure, the value of the implemented function is defined as the rounding of $f(x)$ to the nearest ufix($-1, -4$) number.

This 4-bit format maps the binary number 0000_2 to 0 and the binary number 1111 to $\frac{15}{16}$: It never reaches 1. For the sine function, it is not really a problem on the input side (this will be shown in Chap. 20, dedicated to evaluating the sine and cosine functions—we ask our readers to accept it as true for now). On the output side, however, we do have a problem: $f(x)$ reaches 1 for the two input values $\frac{14}{16}$ and $\frac{15}{16}$. This is indeed a problem, as it forbids the use of the same ufix($-1, -4$) on the output. More precisely, if we do use ufix($-1, -4$) on the output, what will probably happen is that the real value 1 will be mapped to 0 thanks to the modulo property of fixed-point arithmetic: Instead of the largest value, we get the smallest one, which can be considered catastrophic in this context. Furthermore, as it is a corner-case situation, it may remain undetected by random testing or testing on real input data. The main takeaway message of this section is to be aware of this problem and detect it.

Then, when a designer is aware of the problem beforehand, there exist a range of techniques to solve it. We now review a few of them, but the

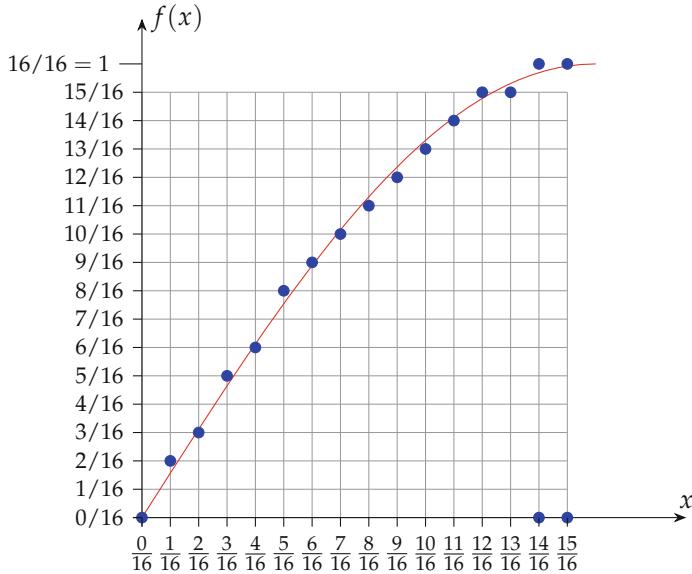


Fig. 16.5 Discretization of $\sin\left(\frac{\pi}{2}x\right)$ on $[0, 1)$, when both input and output domains are mapped on the set $\text{ufix}(-1, -4)$ of 4-bit unsigned fixed-point numbers.

best solution will depend on the mathematics of the function and on the requirements of the application.

- One option, illustrated by Fig. 16.6a, is to use a larger format on the output, here the 5-bit $\text{ufix}(0, -4)$ format. However, nearly half the output range will be unused, which is not very satisfactory.
- One option, illustrated by Fig. 16.6b, is to use saturated arithmetic, mapping 1 to the largest fixed-point value. It may or may not incur some hardware overhead.
- One option, illustrated by Fig. 16.6c, is to specify the discrete function as the rounding of $f(x)$ to the $\text{ufix}(0, \ell)$ number immediately below $f(x)$. In other words, we round down instead of rounding to the nearest.
- One option, illustrated by Fig. 16.6d, consists in scaling the function to map the desired output interval. This is a very generic solution, all the more as it is also possible to add a constant offset. However, it is only acceptable if the application context can tolerate it or be adapted to it. The needed scale factor in our example is very close to 1 ($15/16$ here, $1 - 2^\ell$ in general). A variant of this option (not applicable to the sine) is to apply some scaling or shifting to the input, in such a way as to avoid the problems on the output.

None of these options is ideal. All but the first incur a larger worst-case error, for instance, the saturation option has a worst-case error of $1/32$, or one

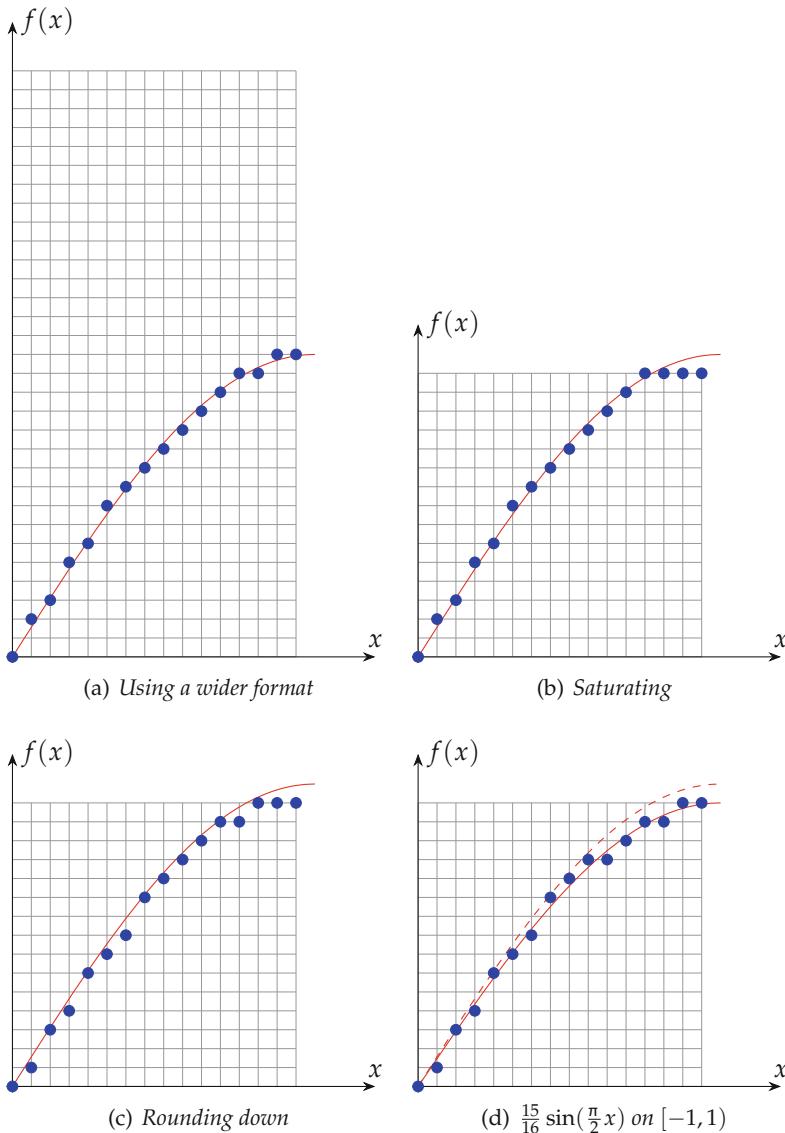


Fig. 16.6 Possible fixes for corner-case discretization issues.

half-ulp, on most of the domain, but for the two rightmost values the error is close to 1/16: one full ulp. Only the last option is still perfectly rounded, but with respect to a modified function, which adds up to one ulp of error with respect to the original function: The overall error bound, with respect to the original function, is now close to 1.5 ulp.

In practice, the main concern of a designer should be to avoid the catastrophic issue of replacing one extremal value of the format by the other. The loss of accuracy is comparatively a minor problem. Besides it can always be taken into account in the application's error analysis (see Chap. 3).

A final subtlety is illustrated by Fig. 16.7 on the example of the reciprocal function $1/x$ on $[1, 2]$ (a typical domain for this function, as larger or smaller values are easily reduced to this domain by the trivial formula $\frac{1}{2^n x} = 2^{-n} \frac{1}{x}$). To obtain a function on $[0, 1)$ we simply consider $\frac{1}{x+1}$ on $[0, 1)$. We further offset it so that the output domain also closely matches a fixed-point format: The function which maps 0 to 1 and 1 to 0 is $f(x) = \frac{2}{x+1} - 1$. Here, from a hardware point of view, both the +1 and -1 are trivial bit operations that entail no cost. However, the issue is that the image of 0 is exactly 1; therefore, the output format requires the bit of weight 0, which is used only for this specific output value $f(0) = 1$.

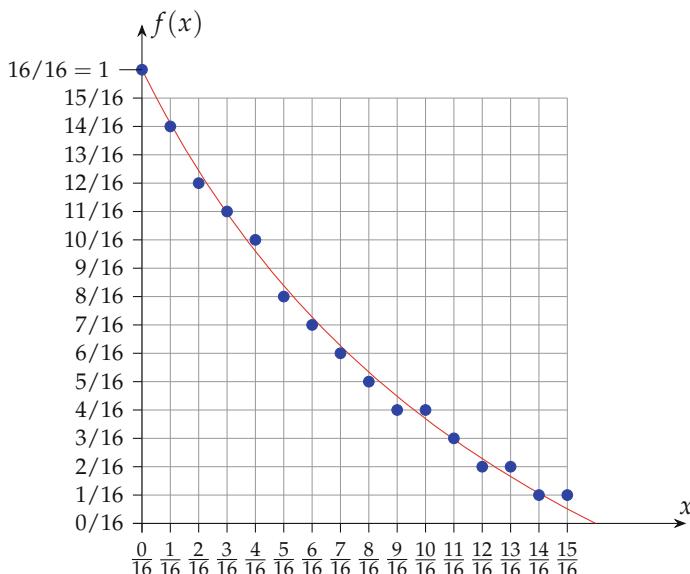


Fig. 16.7 Discretization of $\frac{2}{x+1} - 1$ on $[0, 1)$, when both input and output domains are mapped on the set $\text{ufix}(-1, -4)$ of 4-bit unsigned fixed-point numbers.

This is, again, a small problem in practice, in the sense that a sensible workaround exists for each context where the function is used. However, we again warn against possible catastrophic effects if this issue is ignored.

We have illustrated this section with round to the nearest. For faithful approximators, such discretization issue is even more likely to arise. Figure 16.8 shows, for each value of X, the two possible values that can be returned by a faithful approximator to $\sin(\frac{\pi}{2}x)$.

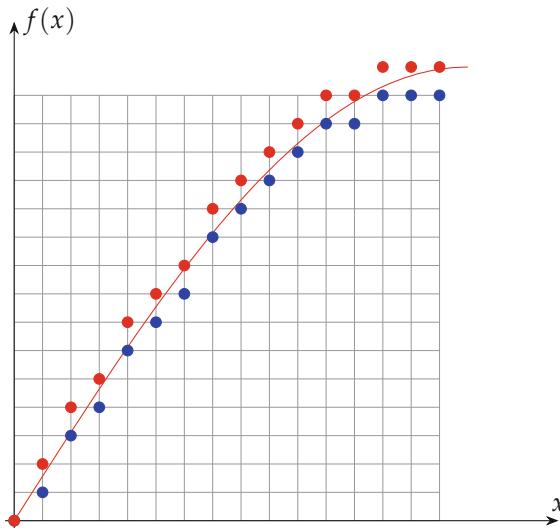


Fig. 16.8 The possible values returned by a faithful approximator to $\sin(\frac{\pi}{2}x)$ on $[-1, 1]$.

This figure allows us to introduce a new discretization issue, probably much less critical in general than the previous range issues: the preservation of the monotonicity (if any) of the function.

16.2.2 Monotonicity Issue

A property of most rounding functions (rounding to nearest, rounding up, and rounding down) is that they are monotonic. Therefore, when applied to a monotonic function f , they preserve this monotonicity. In particular, the round to nearest of a monotonic function presents the same monotonicity property as the original function (see Fig. 16.6).

Conversely, a faithful operator will not preserve the monotonicity of the function it implements. For instance, for the three values to the right of Fig. 16.8, the return value could go up and down. Such non-monotonicities will, by definition, never be larger than one ulp for a faithful operator.

There are few application contexts where it is a problem. To our knowledge, the only published study that considers this issue is an article by Iordache and Matula [IM99] that will be reviewed in its context in Sect. 17.2.10, p. 521.

16.3 Some Classes of Numerical Functions

Functions such as square root and cosine may seem very similar when one only considers their computer incarnation: In the mathematical library, they have similar prototypes (e.g., `sqrt` and `cos` respectively), and even their performance is quite comparable. In the mathematical world, both functions input and output real numbers, and both can output irrational numbers on trivial inputs (e.g., $\sqrt{2}$ or $\cos(\frac{1}{2})$). Yet these two functions belong to very different classes, with a deep impact on the techniques that can be used to evaluate them. This section attempts to overview these classes.

16.3.1 Algebraic Functions

An algebraic function is defined as the solution to a polynomial equation of several variables. For instance:

- The quotient is the solution Q to the equation $P(Q, X, Y) = 0$ where $P(Q, X, Y) = QY - X$.
- The square root S is the positive solution to the equation $P(S, X) = 0$ where $P(S, X) = S^2 - X$.
- The 2D norm $\sqrt{X^2 + Y^2}$ is the positive solution N to the equation $P(N, X, Y) = 0$ where $P(N, X, Y) = N^2 - X^2 - Y^2$,
- The value of $\frac{X}{\sqrt{X^2 + Y^2}}$ is the solution S to the equation $P(S, X, Y) = 0$ where $P(S, X, Y) = S^2(X^2 + Y^2) - X$.

Note that the basic operations (+, -, \times , and $/$) are also bivariate algebraic functions.

Since the polynomial P is defined as a combination of additions and multiplications, it is usually possible to directly derive, from the definition of an algebraic function, an algorithm that evaluates it using only additions and multiplications. The digit recurrence algorithms for division studied in Chap. 9 are an illustration; similar algorithms can be derived for square root [EL94] but also more complex functions such as a 3D norm [TK00] or arbitrary rational functions using the E-method [Erc77; Bri+08; Bri+18]—see Chap. 19.

Furthermore, on inputs that are finite binary numbers, the exact evaluation of the polynomial P is possible and yields a finite result. It may not be possible to achieve exactly $P(S, X, Y, \dots) = 0$ when all the variables of the polynomial are constrained to finite binary representations, but in this case the value of $P(S, X, Y, \dots)$ provides a measure of the error achieved that is finite and computable only with additions and multiplications. An example already encountered in this book is the Euclidean division: Even when the quotient is not representable in binary (e.g., $1/3$), it is possible to compute

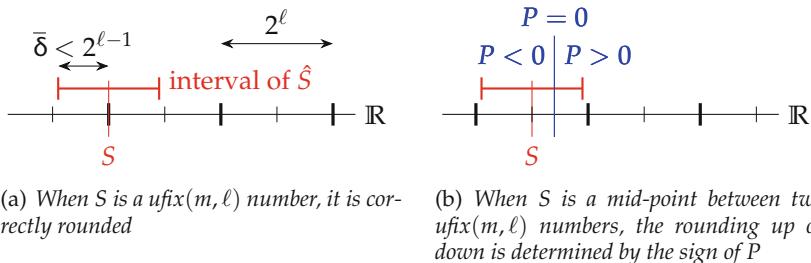


Fig. 16.9 For an algebraic function defined by $P = 0$, faithful rounding to $S \in \text{ufix}(0, \ell - 1)$ enables correct rounding to the nearest ufix($0, \ell$) number.

an approximation to arbitrary precision and a remainder that is in this case the value of $P(Q, X, Y) = QY - X$.

Another practical consequence of the definition of algebraic functions is the possibility to deduce correct rounding to the nearest from a faithful approximation. This technique has been introduced for division and square root (see Sect. 11.3.4, p. 351). Its generalization is illustrated by Fig. 16.9. The exact solution \hat{S} may not be exactly representable, but we assume here that an approximation S , faithful to precision $\ell - 1$, has been computed—this is in general cheaper than aiming directly for the correctly rounded result, for reasons exposed in Chap. 3. Then there are only two cases to consider:

- Either the LSB of S is 0, which means that S is a precision- ℓ number (Fig. 16.9a). In this case, it is necessarily correctly rounded to the nearest at precision ℓ .
- Or S is a midpoint between two precision- ℓ numbers (Fig. 16.9b). As long as $P(S, X, Y, \dots)$ is monotonic in S , it is possible to deduce from the sign of $P(S, X, Y, \dots)$ if $S > \hat{S}$, $S < \hat{S}$, or $S = \hat{S}$. The key point here is that the sign of $P(S, X, Y, \dots)$ can be computed exactly with finite resources, since this expression only involves additions and multiplications of numbers that all have a finite binary representation.

The next chapters will introduce several generic techniques for faithful approximation. The takeout message here is that for algebraic functions, any of these techniques can be converted into a technique for correct rounding.

16.3.2 Elementary and Special Functions

Elementary functions include the exponential and logarithm, the trigonometric functions (which are related to the exponential in the complex domain) and their inverse, and in general any finite composition of these func-

tions with algebraic functions. There are several techniques for evaluating an elementary function with arbitrary accuracy [Mul16].

The class of special functions is (less formally) defined as the class of the useful mathematical functions that are not elementary. A special function is typically defined by an integral or infinite series. The C standard mathematical library includes a few special functions such as the error function (erf) or the Bessel function. They are, in general, more difficult to evaluate than elementary functions, and research is still active on this domain.

Leaving some details aside, a *D-finite* or *holonomic* function is a function that is a solution of a finite system of linear differential equations with polynomial coefficients. The class of D-finite functions includes all the algebraic functions, many (but not all) elementary functions, and many useful special functions. Many properties and algorithms can be automatically deduced from such a definition [Ben+10].

From the point of view of this book, elementary and special functions are fundamentally more difficult than algebraic ones in one respect: achieving correct rounding. There exists a generic algorithm to compute the correct rounding by successive increases of accuracy [Ziv91]. A first problem is to prove its termination on all inputs: If the input X is such that the exact value $f(X)$ is a midpoint, this algorithm enters an infinite loop. Such cases need therefore to be filtered, which may be easy (for instance, the exponential e^x of a nonzero rational x is always irrational, hence not a midpoint) or more difficult [Mul16]. A second problem is that this algorithm, a “while” loop, makes sense in software, but much less in hardware.

This issue is mostly out of the scope of this book, and we refer the reader to Muller’s textbook [Mul16] for more details. In this book, we will only consider correct rounding of elementary functions for small precisions (up to 24 bits) where the construction of the architecture can rely on an exhaustive enumeration of all the correctly rounded values using multiple-precision software. This is the case of the plain tabulation technique introduced next.

16.4 A First Generic Approximator: Simple Tabulation

We conclude this chapter with the simplest technique providing a generic function approximator: plain tabulation. It is also the technique that has the lightest requirements on the function to be approximated: It only needs the function to be defined (with a finite value) on its domain. In contrast, most techniques studied in later chapters require some form of differentiability of the function on its interval. Also, there is no architectural overhead in a correctly rounded table (the overhead is only in the code that fills the table, but it is of little practical significance).

The interface to the corresponding generator, shown in Fig. 16.10, is also very simple.

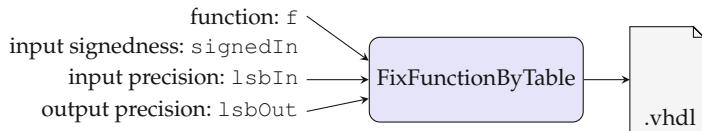


Fig. 16.10 Parameters of `FixFunctionByTable`.

16.4.1 Tabulating Precomputed Function Values in a ROM

The architecture is then a simple read-only memory (ROM) that is filled with the values of the function.

These values are computed, using arbitrary precision and arbitrarily complex algorithms, at architecture-generation time. In FloPoCo, this complexity is delegated to the Sollya tool [CJL10]: The function is input as an arbitrary Sollya expression, examples of which will be given in this chapter and the following ones. One advantage of using Sollya is that it supports many elementary functions—essentially all the functions implemented in MPFR⁴ [Fou+07]. However, its main killer feature in the present context is that it offers to evaluate these expressions with guaranteed accuracy, even if they involve potentially accuracy-degrading features such as cancellations. This process is rather involved [CJL10], but is hidden to FloPoCo users and developers.

Here are a few examples of how to use FloPoCo to produce tables of precalculated function values, using the operator `FixFunctionByTable`. They illustrate some of the issues discussed in Sect. 16.2.

Hands on: Plain tabulation using `FixFunctionByTable`



- 1) An example without range issue: $\sin(\frac{\pi}{4}x)$ on $[0, 1]$

In this case, the image of $[0, 1]$ by the function is included in $[0, 1]$, so the resulting table will have the same input and output formats if we provide the same `lsbOut`. The command to use is

```
flopoco FixFunctionByTable f="sin(pi/4*x)" \
signedIn=false lsbIn=-8 lsbOut=-8
```



- 2) An example where the output reaches 1: $\sin(\frac{\pi}{2}x)$ on $[0, 1]$
A naive approach would be:

⁴ <http://www.mpfr.org>.

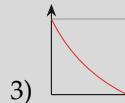
```
flopoco FixFunctionByTable f="sin(pi/2*x)" \
    signedIn=false lsbIn=-8 lsbOut=-8
```

Looking at the resulting VHDL shows that the output includes one MSB bit that is only useful to represent the value 1: Almost half of the output domain is unused. Besides, we now have an 8-bit in, 9-bit out table.

Another solution, which may be better in some contexts, is to scale the output of the function so that it maps to the same domain as the input. Here, the proper scaling factor is $\frac{255}{256}$:

```
flopoco FixFunctionByTable f="255/256*sin(pi/2*x)" \
    signedIn=false lsbIn=-8 lsbOut=-8
```

The scaling factor can also be expressed in Sollya syntax as $(1b8-1)/1b8$, where b means “times two to the power.” The reader may check that the resulting VHDL maps 00000000 to 00000000, and 11111111 to 11111111.

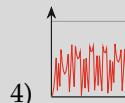


3) When $[0,1]$ is mapped to $(0,1)$: $\frac{2}{1+x} - 1$ on $[0, 1]$

This function is of practical use: It can be wrapped in leading zero counters and shifters to implement $1/x$ on a larger fixed-point domain. In such uses, it is not a problem to have an extra MSB bit, as this range reduction/reconstruction logic will consume the extra bit:

```
flopoco FixFunctionByTable f="2/(x+1)-1" signedIn=false \
    lsbIn=-8 lsbOut=-16
```

This example also illustrates that the output precision may be much larger than the input precision: The size of the VHDL is exponential in the input size, but only linear in the output size.



4) Example of tabulation in an integer context: $x^2 \bmod p$

This function, when x is an integer, can be used to implement, by tabulation, the modular multiplication $a \times b \bmod p$, using the modulo p version of $a \times b = \frac{(a+b)^2 - (a-b)^2}{4}$. All the data here is integer, so the expression is a bit convoluted. In the following example, the 2^6 possible inputs in $[0,1)$ are first scaled to the first 2^6 consecutive integers by multiplication by 2^6 , denoted as $1b6$ in Sollya syntax. The modulo operation also has to be implemented out of the floor function:

```
flopoco FixFunctionByTable \
f="x*1b6*x*1b6-23*floor(x*1b6*x*1b6/23)" \
    signedIn=false lsbIn=-6 lsbOut=0
```

The reader is invited to check that the table in the resulting VHDL implements the desired function.

The mathematical function implemented here has many discontinuities: This example also illustrates that tabulation only requires the function to be defined on its domain.

16.4.2 Actual Cost of a ROM

A table of k input bits and n output bits is often described as a $2^k \times n$ ROM. Conceptually, it holds $2^k \times n$ bits, and for example, the source VHDL files obtained from the above FloPoCo invocations contain arrays of 2^k lines, each of n zeroes or ones. However, $2^k \times n$ does not accurately describe the actual cost of a tabulated function. The actual cost strongly depends on the technology used to actually implement the table.

On ASIC, small tables are best considered as Boolean functions and implemented as standard logic, thus benefiting from the very powerful logic optimization capabilities of modern tools. This has been studied empirically by Gustafsson and Johansson [GJ08], who found that the area A of such a solution is quite symmetrical in k and n : They report

$$A \propto 2^{0.65 \min(k,n)} \times 2^{0.19|k-n|}. \quad (16.1)$$

The reason is that tools not only attempt to minimize the logical expression associated to each output; they also attempt to maximize sub-expression sharing between outputs. Thus, a graph of gates transforming k input to n outputs has roughly the same cost as a graph of gates transforming n inputs to k outputs.

Another option on ASIC is to use ROM generators. These may benefit from a layout optimized at the transistor level and should have a very low cost per bit: In a ROM the logic cost is essentially in the address-decoding logic, since constant 0 and 1 resume to wiring to the corresponding power lines. However, ROMs also have the drawback of less layout flexibility, and they do not benefit from the aforementioned logic optimization. In [GJ08] ROM generators were found to lead to larger area than standard-cell implementations on the tabulated functions studied.

On FPGAs, we have seen in Sect. 4.3 that ROMs can be built out of two primitives. The first primitive is a small LUTs with α input bits and one output bit, so one LUT can be viewed as a $2^\alpha \times 1$ ROM. A $2^\alpha \times n$ -bit ROM can be built by assembling n LUTs, as shown in Fig. 16.11.

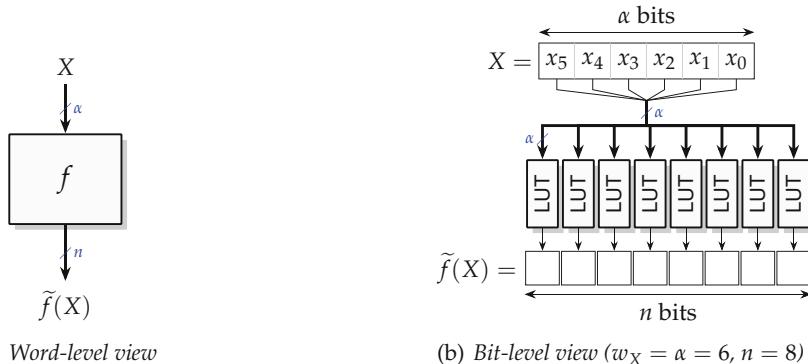


Fig. 16.11 Tabulating a function using FPGA LUTs.

The LUT defines the unit cost on FPGAs: A $2^\alpha \times n$ -bit ROM costs n LUTs, but a $2^{\alpha-1} \times n$ -bit one also costs n LUTs, the latter only partially used.

For $w_X = \alpha + 1$, each output bit will cost 2 LUTs, plus one address-decoding multiplexer (which may be available for free in the FPGA fabric, or not). In general, a $2^k \times n$ ROM should cost $2^{\max(k-\alpha, 0)} \times n$ LUTs.

However, this formula is no more reliable than $2^k \times n$ for ASICs. FPGA synthesis tools also inherit from the powerful logic optimization techniques developed for ASICs and will typically be able to synthesize a ROM in fewer LUTs than $2^k \times n$. For instance, on an FPGA with $\alpha = 6$, the $2^8 \times 8$ table generated for $\sin(\frac{\pi}{2}x)$ on $[0, 1]$ is synthesized in 25 LUTs instead of the predicted $2^2 \times 8 = 32$.

The second primitive available is the large block RAM (which can also be used as ROM, its content being loaded at configuration time). It has a very discrete output size (see Table 4.2), and the same lack of flexibility as an ASIC ROM, but offers a relevant solution for tables of a few tens of kilobits.

References

- [AC11] Mark G. Arnold and Caroline Collange. “A Real/Complex Logarithmic Number System ALU”. In: *IEEE Transactions on Computers* 60.2 (2011), pp. 202–213 (cit. on p. 478).
- [AS64] Milton Abramowitz and Irene A. Stegun. *Handbook of mathematical functions*. Applied Math. Series 55. National Bureau of Standards, Washington, D.C., 1964 (cit. on p. 478).
- [Ben+10] Alexandre Benoit, Frédéric Chyzak, Alexis Darrasse, Stefan Gerhold, Marc Mezzarobba, and Bruno Salvy. “The dynamic dictionary of mathematical functions (DDMF)”. In: *International*

- Congress on Mathematical Software. Springer. 2010, pp. 35–41 (cit. on p. 488).
- [Bri+08] Nicolas Brisebarre, Sylvain Chevillard, Miloš D. Ercegovac, Jean-Michel Muller, and Serge Torres. “An Efficient Method for Evaluating Polynomial and Rational Function Approximations”. In: *International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE, 2008, pp. 245–250 (cit. on p. 486).
- [Bri+18] Nicolas Brisebarre, George Constantinides, Miloš Ercegovac, Silviu-Ioan Filip, Matei Istoan, and Jean-Michel Muller. “A High Throughput Polynomial and Rational Function Approximations Evaluator”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2018, pp. 99–106 (cit. on p. 486).
- [Che+07] Ray C.C. Cheung, Dong-U. Lee, Wayne Luk, and John D. Villasenor. “Hardware Generation of Arbitrary Random Number Distributions From Uniform Distributions Via the Inversion Method”. In: *IEEE Transactions on VLSI Systems* 8.15 (2007) (cit. on p. 477).
- [CHT02] Marius Cornea, John Harrison, and Ping Tak Peter Tang. *Scientific Computing on Itanium®-Based Systems*. Intel Press, 2002 (cit. on p. 479).
- [CJL10] Sylvain Chevillard, Mioara Joldes, and Christoph Lauter. “Sollya: An Environment for the Development of Numerical Codes”. In: *Mathematical Software - ICMS 2010*. Vol. 6327. Lecture Notes in Computer Science. Springer, 2010, pp. 28–31 (cit. on p. 489).
- [DD07a] Jérémie Detrey and Florent de Dinechin. “Floating-Point Trigonometric Functions for FPGAs”. In: *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2007, pp. 29–34 (cit. on p. 477).
- [DD07b] Jérémie Detrey and Florent de Dinechin. “Parameterized floating-point logarithm and exponential functions for FPGAs”. In: *Microprocessors and Microsystems, Special Issue on FPGA-based Reconfigurable Computing* 31.8 (2007), pp. 537–545 (cit. on p. 477).
- [DT05] Florent de Dinechin and Arnaud Tisserand. “Multipartite Table Methods”. In: *IEEE Transactions on Computers* 54.3 (2005), pp. 319–330 (cit. on p. 479).
- [EL94] Miloš D. Ercegovac and Tomás Lang. *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, Boston, 1994 (cit. on p. 486).
- [Erc77] Miloš D. Ercegovac. “A General Hardware-Oriented Method for Evaluation of Functions and Computations in a Digital Computer”. In: *IEEE Transactions on Computers* C-26.7 (1977), pp. 667–680 (cit. on p. 486).

- [Fou+07] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. "MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding". In: *ACM Transactions on Mathematical Software* 33.2 (2007), 13:1–13:15 (cit. on p. 489).
- [GJ08] Oscar Gustafsson and Kenny Johansson. "An Empirical Study on Standard Cell Synthesis of Elementary Function Lookup Tables". In: *Asilomar Conference on Signals, Circuits and Systems*. IEEE, 2008, pp. 1810–1813 (cit. on p. 491).
- [IM99] Cristina Iordache and David W. Matula. "Analysis of Reciprocal and Square Root Reciprocal Instructions in the AMD K6-2 Implementation of 3DNow!". In: *Electronic Notes in Theoretical Computer Science* 24 (1999) (cit. on p. 485).
- [KD09] Nachiket Kapre and Andre DeHon. "Accelerating SPICE Model-Evaluation using FPGAs". In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2009, pp. 37–44 (cit. on p. 478).
- [Lee+05] Dong-U Lee, Altaf A. Gaffar, Oskar Mencer, and Wayne Luk. "Optimizing Hardware Function Evaluation". In: *IEEE Transactions on Computers* 54.12 (2005), pp. 1520–1531 (cit. on p. 479).
- [Lee+06] Dong-U Lee, John D. Villasenor, Wayne Luk, and Philip H.W. Leong. "A Hardware Gaussian Noise Generator Using the Box-Muller Method and Its Error Analysis". In: *IEEE Transactions on Computers* 55.6 (2006) (cit. on p. 477).
- [Mar00] Peter Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000 (cit. on pp. 477, 479).
- [Meh+09] Pramod K. Meher, Javier Valls, Tso-Bing Juang, K. Sridharan, and Koushik Maharatna. "50 Years of CORDIC: Algorithms, Architectures, and Applications". In: *IEEE Transactions on Circuits and Systems I: Regular papers* 56.9 (2009), pp. 1893–1907 (cit. on p. 478).
- [Mul+18] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. 2nd ed. Birkhäuser Boston, 2018 (cit. on p. 479).
- [Mul16] Jean-Michel Muller. *Elementary Functions, Algorithms and Implementation*. 3rd ed. Birkhäuser Boston, 2016 (cit. on pp. 478, 479, 488).
- [Tho15] David B. Thomas. "A general-purpose method for faithfully rounded floating-point function approximation in FPGAs". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2015 (cit. on p. 478).

- [TK00] Naofumi Takagi and Seiji Kuwahara. "A VLSI Algorithm for Computing the Euclidean Norm of a 3D Vector". In: *IEEE Transactions on Computers* 49.10 (2000), pp. 1074–1082 (cit. on p. [486](#)).
- [VCA10] Panagiotis D. Vouzis, Caroline Collange, and Mark G. Arnold. "A Novel Cotransformation for LNS Subtraction". In: *Journal of Signal Processing Systems* 58.1 (2010), pp. 29–40 (cit. on p. [478](#)).
- [Ziv91] Abraham Ziv. "Fast evaluation of elementary mathematical functions with correctly rounded last bit". In: *ACM Transactions on Mathematical Software* 17.3 (1991), pp. 410–423 (cit. on p. [488](#)).



17

CHAPTER 17

Function Evaluation Using Tables and Additions

This chapter surveys a range of generic function approximation methods that primarily rely on tables of precomputed values and additions. The methods themselves may be complex but the resulting hardware is simple. Compared to plain tables, these methods trade area for latency.

It is surprising what one can achieve by simply adding precomputed values. Section 17.1 presents a generic method that replaces one large table with two smaller tables and one addition. The table size is typically reduced by half, and the overhead of the addition is very low. Section 17.2 presents a family of piecewise linear approximation methods where the products are themselves tabulated to avoid the need for multipliers. This method can reduce the table size by one order of magnitude compared to plain tabulation, but the overhead of the adders is no longer negligible. Section 17.3 overviews other table-and-addition methods that use additions before and after the tables.

17.1 Lossless Differential Table Compression

Lossless differential table compression (LDTC) replaces one table of values with two smaller tables and an addition (Fig. 17.1), such that the result of this addition is the exact same value as the output of the original table (*errorless* or *lossless* compression). This method was introduced and then improved by Hsiao et al. [Hsi+15; Hsi+17] in the specific context of the multipartite table method that is the subject of Sect. 17.2. It was later generalized to other contexts [CFD22] and is indeed generic enough that it can be presented first.

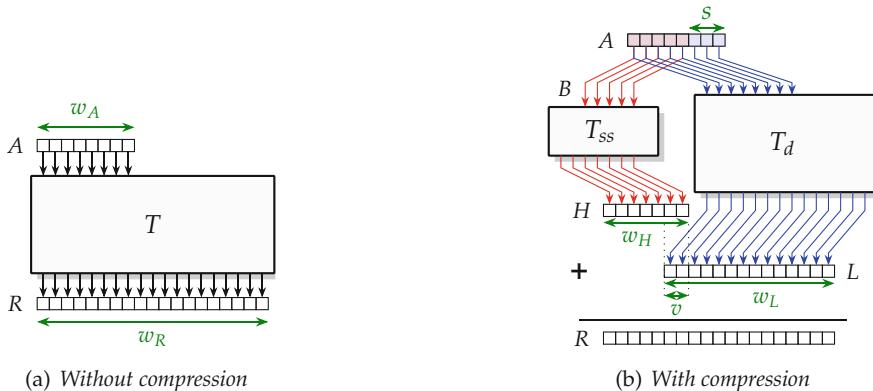


Fig. 17.1 Table compression example (here for the TIV of a multipartite architecture for $\sin(\frac{\pi}{4}x)$ on $[0, 1]$ with 16-bit inputs and outputs).

17.1.1 Applicability

Figure 17.1a shows an uncompressed table T with $w_A = 8$ address bits and $w_R = 19$ output bits. Informally, a table T has some potential for lossless differential table compression (LDTC) if the values stored at neighboring addresses present small variations with respect to the full output range of the table. This property is often satisfied when tabulating a function as per the previous chapter, at least as long as the function is regular enough (i.e., continuous and differentiable with bounded derivatives on the interval of interest). However, what is important for LDTC is not these mathematical properties of the underlying real-valued function, but the “small local variations” property of the discrete table. For illustration, Fig. 18.13, p. 546, plots the content of four discrete tables that are the result of a numerical optimization process [BC07]. There is no closed-form real function of which any of these tables is a sampling, the content of the C_3 table of this figure is not even monotonic, yet these tables are perfectly suitable for the compression studied here.

To our readers who rightly feel that this “small local variations” property is defined too informally, the best formal definition would be: A table T can be compressed by LDTC if Algorithm 17.1 (presented below) succeeds. Unfortunately, this definition is even less intuitive.

LDTC can be used (among others) in three of the subsequent chapters of the present book:

- to compress polynomial coefficient tables (such as the already mentioned Fig. 18.13) in Chap. 18,
- to compress the sine/cosine table of the architecture presented in Fig. 20.11 of Chap. 20,

Table 17.1 Notations used in this section.

Original table	$T : A \mapsto R$
Input and output sizes	w_A, w_R
Subsampling table	$T_{ss} : B \mapsto H$
Input and output sizes	$w_B = w_A - s, w_H$
Difference table	$T_d : A \mapsto L$
Input and output sizes	w_A, w_L
Number of overlap bits	$v = w_H + w_L - w_R$

- to compress the exp table of the architecture presented in Fig. 22.6 of Chap. 22.

The tables in the KCM constant multiplier architectures of Chap. 12 would also be a good candidate, but there the compression potential seems to be already exploited by the KCM technique itself.

17.1.2 The Parameter Space of LDTC

The core idea [Hsi+15] of LDTC is the following (with the notations described in Fig. 17.1 and Table 17.1). The original table T is sub-sampled by a factor 2^s , which gives a subsampling table T_{ss} . Obviously, T_{ss} is smaller than T since it has fewer entries ($2^{w_A - s}$ instead of 2^w_A). As shown in Fig. 17.1b, each value of the original table is then reconstructed by adding, to one entry of T_{ss} , the difference between this entry and the original value of T . This difference is stored in a second table T_d of 2^w_A entries. This table T_d has as many entries as the original table T , but its output range w_L is smaller than that of T thanks to the “small local variations” property of T : Indeed T_d stores local variations. Hence, T_d has fewer output bits than the original table and is therefore also smaller. There is a compression as soon as the sum of the sizes of the two smaller tables is smaller than the original size of T [Hsi+15]. Reconstructing the value of T requires an addition, whose architectural cost will be discussed in Sect. 17.1.4.

In all the following, we call a *slice* of T the subset of 2^s consecutive values to be reconstructed from one value of T_{ss} . If built as exposed previously, T_d systematically has 2^s entries equal to 0: one for each slice. These systematic zeroes suggest that a further optimization is possible. Indeed, an improvement in [Hsi+17] is to add, to each entry of T_d , the value of the $w_R - w_L$ least significant bits (LSB) of the corresponding T_{ss} entry. Thus, these bits can be removed from T_{ss} , reducing its output size by $w_R - w_L$ bits. However, we now have larger values in T_d : In some cases this adds one bit (overflow bit) to the output size of T_d .

The possible overflow bit in T_d is expensive, since it is added to 2^{w_A} entries. For this reason, instead of removing the maximum number of LSB output bits from T_{ss} [Hsi+17], it may be interesting to leave some of these bits if it allows us to avoid the overflow bit in T_d . If k extra output bits in T_{ss} allow for a T_d without overflow, the extra cost is $k \times 2^s$ bits and the benefit is 2^w_A bits, so there is a potential net gain in storage.

Conversely, once we acknowledge that T_d may overflow and that its output size w_L must be enlarged, it is worth attempting to reduce w_H at the LSB to benefit from the new freedom that a wider w_L provides.

To capture all these cases, for a given table T with its input size w_A and output size w_R , a compression parameter vector is defined as the triplet (s, w_H, w_L) shown in Fig. 17.1. A vector is valid if it is possible to achieve LDTC with these parameters. A vector also has an implementation cost, estimated thanks to a cost function $c(w_A, s, w_H, w_L)$.

17.1.3 The LDTC Optimization Algorithm

A generic LDTC optimization is then provided by Algorithm 17.1. It simply enumerates this parameter space and selects among the valid vectors the one with the smallest cost. This space is fairly small since s , w_H , and w_L are numbers of bits.

Algorithm 17.1: Generic LDTC optimization

```

function optimizeLDTC( $T, w_A, w_R$ )
     $bestVector \leftarrow (0, w_R, 0)$                                 // no compression
     $bestCost \leftarrow c(bestVector)$ 
    forall  $(s, w_H, w_L)$  do
         $cost \leftarrow c(w_A, s, w_H, w_L)$ 
        if  $cost < bestCost$  then
            if isValid( $T, w_A, w_R, s, w_H, w_L$ ) then
                 $bestCost \leftarrow cost$ 
                 $bestVector \leftarrow (s, w_H, w_L)$ 
            end if
        end if
    end forall
    return  $bestVector$ 

```

Algorithm 17.1 first filters by cost and then by validity, assuming that the cost function is faster to evaluate than validity. Indeed, the validity of a parameter vector is another exhaustive check, described by Algorithm 17.2.

This approach is the most naive but also the most generic, requiring no more hypothesis on the tabulated values than the values themselves.

Algorithm 17.2: Is a parameter vector valid?

```

function isValid( $T, w_A, w_R, s, w_H, w_L$ )
  for  $B \in (0, 1, \dots, 2^{w_A-s} - 1)$                                 // loop on slices
    do
       $S \leftarrow \{T[j]\}_{j \in \{B \cdot 2^s \dots (B+1) \cdot 2^s - 1\}}$           // slice
       $M \leftarrow \max(S)$                                               // max on slice
       $m \leftarrow \min(S)$                                               // min on slice
       $mask \leftarrow 2^{w_R-w_H} - 1$ 
       $H \leftarrow m - (m \& mask)$                                          // w_H upper bits of m
       $M_{low} \leftarrow M - H$                                             // max diff value on this slice
      if  $M_{low} \geq 2^{w_L}$  then
        return false                                                 // one slice won't fit: exit with false
      end if
    end for
  return true

```

Note that Algorithm 17.2 is faster than attempting to fill the tables: It only needs the max and min of T on each slice, which can be computed only once for each value of s and memoized. Therefore one invocation of Algorithm 17.2 requires time proportional to 2^{w_A-s} , not to 2^{w_A} .

Altogether, the exploration of this parameter space on current computers is almost instantaneous for any size for which tabulation is practical.

17.1.4 Cost Function

In terms of area, the cost $c(w_A, s, w_H, w_L)$ of an LDTC-compressed table is the sum of the cost of T_{ss} , the cost of T_d , and the cost of the addition.

The cost of a table depends on the technology used and has been discussed in Sect. 16.4.2 of the previous chapter.

If an adder is used to compute the sum, it should be obvious from Fig. 17.1b that the adder size is only w_H bits (the $w_R - w_H$ lower bits of the sum are those of L).

However, in many of the applications, the table result is added to a value that is itself computed by a compression tree. This is the case in the original application to the multipartite method, where the table to be compressed is the one labeled TIV on Fig. 17.11, p. 511. It is also the case for all the coefficients except C_3 in Fig. 18.13, p. 546, if these tables are used by the architecture shown in Fig. 18.19, p. 555. Many other examples exist [CFD22].

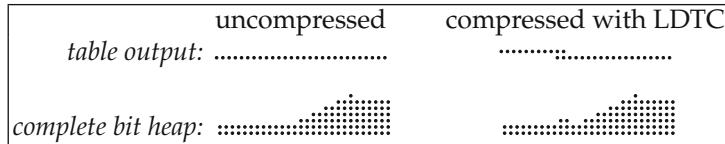


Fig. 17.2 The dot-diagram overhead of lossless table compression, here for the 24-bit multipartite implementation of $\sin(\frac{\pi}{4}x)$.

In such cases, a bit heap may be used (see Chap. 7 of this book), and the addition only adds $v = w_H + w_L - w_R$ bits to this bit heap. This is illustrated in Fig. 17.2. The area cost of the addition therefore becomes proportional to the overlap v , in other words negligible. Furthermore, as long as the addition of these v bits does not entail one more compression stage, the delay overhead will be zero. It is the case in Fig. 17.2.

17.1.5 Evaluation and Observations

LDTC has been integrated in FloPoCo, so that it may benefit all the table-based operators: It can be enabled by the `tableCompression` command-line option. Compression ratios (in bit count, neglecting the addition cost) of 0.5 to 0.8 are common in practical cases (i.e., 20% to 50% savings). The best observed compression ratio was 0.36 [CFD22].

Hands on: FixFunctionByTable using LDTC

The tabulated functions of the previous chapter can often be compressed:

```
flopoco FixFunctionByTable tableCompression=1 \
    f="sin(pi/4*x)" signedIn=false lsbIn=-8 lsbOut=-8
```

This command reports a compression ratio of 0.48 with the following parameters:

Initial cost (T): $8 \times 2^8 = 2048$
Best subsampling cost (T_{ss}): $7 \times 2^5 = 224$
Best diff cost (T_d): $3 \times 2^8 = 768$
Total compressed cost: 992

In general, LDTC works better for tables with the smaller difference between w_R and w_A . The intuition here is that the larger table is always T_d and that T_d has very few output bits when $w_R \approx w_A$. Also, although most applications tend to use tables with $w_R \geq w_A$, LDTC is also effective when $w_R < w_A$. The optimal overlap is often $v = 2$ bits, with a few practical case having their optimal compression for $v = 3$.

17.2 Multipartite Methods

The family of methods we present now has many contributors. The bipartite method was independently invented by Das Sarma and Matula [DM95] in the specific case of the reciprocal function and by Sunderland et al. in the specific case of the sine function [Sun+84]. It was then independently generalized to arbitrary functions by Schulte and Stine [SS99a; SS99b] and Muller [Mul99], both teams contributing different improvements, in particular to use more than two tables [SS99b; Mul99]. These two approaches were merged, and again generalized and improved, by Dinechin and Tisserand [DT05]. Finally, Hsiao showed that one of the tables could be further compressed [Hsi+15; Hsi+17] using the technique of the previous section. The present section is a synthesis of all these works, as implemented in FloPoCo.

17.2.1 Applicability

Contrary to the LDTC technique of the previous section, multipartite methods are fundamentally *approximation* techniques: They address the construction of an architecture that approximates a real-valued function $f(x)$.

For this, these methods require continuity and derivability of the function. In addition, they also need that its derivative remains well bounded. For instance, it should be clear from the start that these methods, for instance, will not work, or will work poorly, on the square root function near zero: As Fig. 17.3a shows, it has a vertical tangent here, which means that its derivative tends to infinity as x comes close to zero.

However, some of these methods were precisely invented to implement square root and reciprocal functions. The trick is to use a function-specific range reduction so that the function is evaluated on a different domain where it is better behaved, for instance, $[1, 2)$ for the square root (see Sect. 19.1, p. 574, for the details of this range reduction).

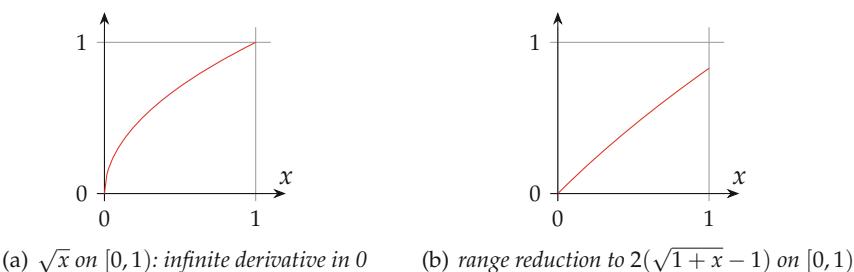


Fig. 17.3 Range reduction to avoid the infinite derivative in the square root function.

For the sake of simplicity, we assume a fixed-point input in $[0, 1)$ in the following. In the square root example, a simple change of variable transforms \sqrt{x} on $[1, 2]$ into $2(\sqrt{1+x} - 1)$ which is plotted in Fig. 17.3b.

17.2.2 The Basic Bipartite Method

A first important idea, illustrated in Fig. 17.4, is to decompose the input word X into its α most significant bits (which form the word A) and its β least significant bits (which form the word B).

Figure 17.5 shows the graph of a function f , with the discretization of its input and output. Each value of A correspond to a sub-domain of the input range containing 2^β values. On this figure, $\alpha = 4$ and $\beta = 2$, so there are $2^4 = 16$ sub-domains, delimited by the vertical red lines, and each containing $2^\beta = 4$ points.

A second idea is to approximate the function by a piecewise linear one: On each of these 2^α sub-domains, the function is replaced by the linear segment that best approximates it. In Fig. 17.5, this approximation is very close, visually indistinguishable from the curve. In more formal terms, the approximation error (the vertical difference between the graph of the function and the graph of the linear approximation) is, on each point of the domain, negligible compared to the output discretization.

If we wanted to build an architecture based on this approximation, we would need to tabulate two values: one initial value $f(A)$ (the red cross in the figure) and the slope $s(A)$ of the segment. Then intermediate values (the blue crosses on the figure) could be interpolated by computing $f(X) = f(A + 2^{-\alpha}B) \approx f(A) + s(A) \times 2^{-\alpha}B$. This is a particular case of *uniform piecewise polynomial approximation* that will be studied in more detail in the next chapter.

The core idea behind the bipartite method is to introduce a degradation of this (too good) approximation that will enable saving the multiplier. Figure 17.6 illustrates this new approximation. It consists in grouping the 2^α input intervals into 2^γ (with $\gamma < \alpha$) larger intervals ($2^2 = 4$ intervals in Fig. 17.6, delimited by the green lines) such that the slope of the segments is considered constant on each larger interval. In other words the bipartite

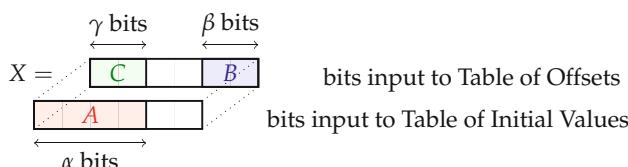


Fig. 17.4 Input word decomposition in the bipartite method.

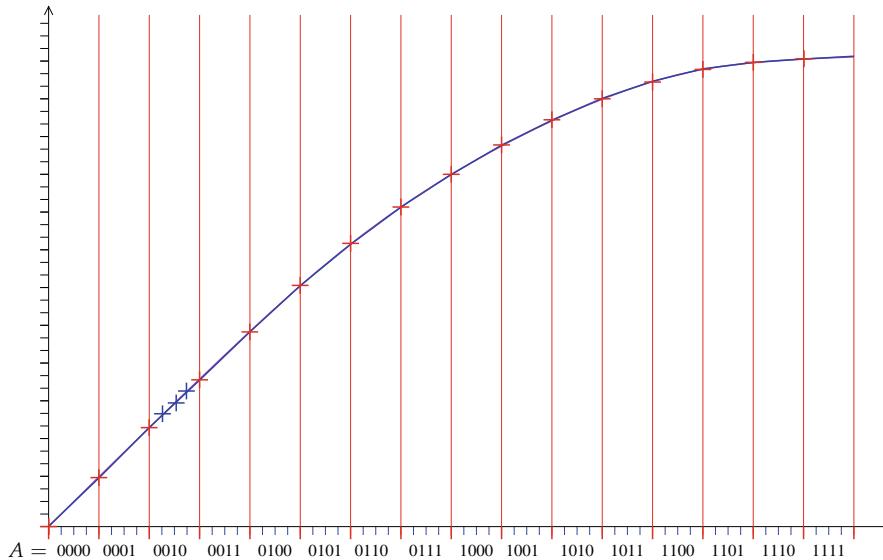


Fig. 17.5 Piecewise linear approximation to a function.

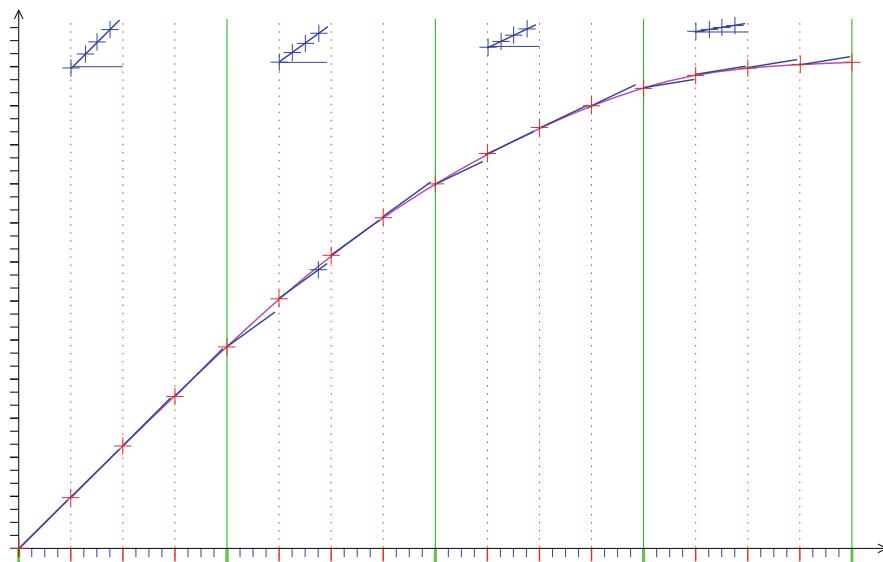


Fig. 17.6 The bipartite approximation ($\alpha = 4$, $\gamma = 2$).

approximation computes $f(A + 2^{-\alpha}B) \approx f(A) + s(C) \times 2^{-\alpha}B$ where C is a subword of A (see Fig. 17.4). These 4 constant slopes are figured at the top of Fig. 17.6, and the corresponding offsets $s(C) \times 2^{-\alpha}B$ (the blue crosses) may be tabulated instead of being computed by a multiplier. There will only be 2^γ

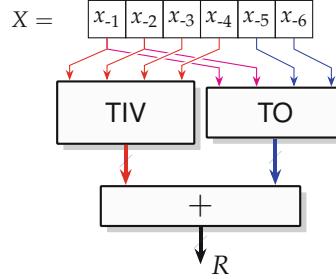


Fig. 17.7 Example bipartite abstract architecture ($\alpha = 4, \beta = 2, \gamma = 2$).

tables of offsets, each containing 2^β offsets. All these tables may be grouped in a single one, addressed by the concatenation of the sub-words C and B of the input.

Figure 17.7 shows the abstract architecture corresponding to this approximation. It is abstract in the sense that we ignore so far the issue of output discretization – this will be managed in the sequel.

This architecture uses two tables, one of 2^α values and one of $2^{\gamma+\beta}$ values. In our (small) example, we need two tables of 16 values each. Note that if we had used a multiplier to compute the offset, we would also have needed two tables of 16 values each (one for initial values, one for slopes). We would also need the adder of Fig. 17.7 to add the product. On this example, the multipartite idea seems a pure improvement. Of course a proper evaluation would need to consider not only the number of table entries, but also the size of each entry – this will also be managed in the sequel.

Compared to a plain table (as per Sect. 16.4 of the previous chapter) which would store $2^{w_X} = 2^{\alpha+\beta}$ values (64 in our example), we have halved the number of tables entries, at the cost of an addition.

In all the following, we will call the *table of initial values* (TIV) the table that stores the initial points of each segment (the red crosses). This table will be addressed by a sub-word A of the input word, made of the α most significant bits. A *table of offsets* (TO) will be addressed by the concatenation of two sub-words of the input word: C (the γ most significant bits) and B (the β least significant bits).

Previous authors [Mul99; SS99b] have expressed the bipartite idea in terms of a Taylor approximation, which allows for a mathematical analysis of the approximation error. They find that for $\gamma \approx \beta \approx w_X/3$ and $\alpha \approx 2w_X/3$, it is possible to keep the error entailed by this method in “acceptable bounds” (the error obviously depends on the function under consideration). We develop below a more geometrical approach to the error analysis, with the purpose of computing the approximation error exactly, where Taylor formulas only give upper bounds. However, before that, we introduce several refinements to the bipartite approximation.

17.2.3 Exploiting Symmetry

First, Schulte and Stine remarked [SS99a] that each linear approximation segment is (trivially) symmetric with respect to its center. Figure 17.8 (left) is a zoom view of Fig. 17.6 when exploiting this idea. In this figure, the TIV now stores the value of the function in the middle of the small interval, and the TO only stores the offsets for a half-segment. On the other half, the values are computed by symmetry.

This technique halves the size of the TO, at the cost of hardware computing the symmetry. In two's complement, the extra hardware consists of a few XOR gates, and its cost is usually more than compensated by the reduction in the TO size [SS99a]. These gates are shown in Fig. 17.8 (right).

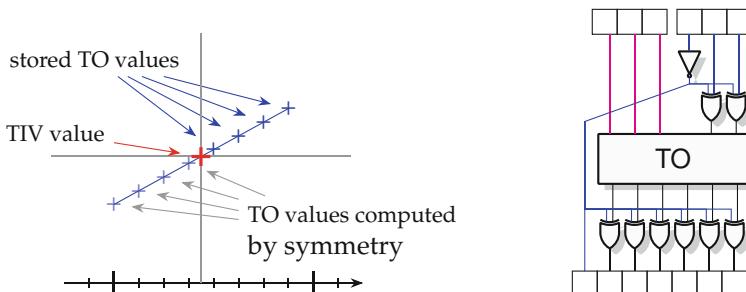


Fig. 17.8 Using symmetry to trade one table input bit for two rows of XOR gates ($\beta = 3, \gamma = 3$).

The initial bipartite paper [DM95] suggested using an “average curve” approximation instead of a linear one for the TO – indeed, since the values are tabulated, they do not need to be strictly aligned. This idea would not improve the maximum error, but would bring a small improvement to the average error. However, in this case Fig. 17.8 would no longer be symmetric, so we could not halve the TO size. A halving of the TO size seems more desirable than this small accuracy gain; therefore, this idea will not be considered further.

Hands on: Observing bipartite tables

The following command will build a bipartite architecture that more or less matches the figures observed so far.

```
flopoco FixFunctionByMultipartiteTable \
  nbTO=1 f="63/64*sin(pi/2*x)" signedIn=false lsbIn=-6
  lsbOut=-6
```

The tables are at the beginning of the VHDL. The reader is invited to observe the values stored in each of them and then look for the XORs that implement the symmetry. The remainder of the architecture is a bit convoluted because of the bit heap framework. Observe how the size of the tables evolve when changing the values of the `1sbIn` and `1sbOut` parameters.

17.2.4 From Bipartite to Multipartite: Splitting the TO

Schulte and Stine [SS99b] and Muller [Mul99] also independently remarked that the TO could be decomposed into several smaller tables. Indeed, what the TO computes is a linear function $\text{TO}(C, B) \approx s(C) \times B$ where $s(C)$ is the slope of the segment. The sub-word B can be decomposed (as seen in Fig. 17.9) into m sub-words B_i of sizes β_i for $0 \leq i < m$:

$$B = B_0 + 2^{\beta_0} B_1 + \dots + 2^{\beta_0+\beta_1+\dots+\beta_{m-2}} B_{m-1}. \quad (17.1)$$

Thus, the TO can be distributed into m smaller tables $\text{TO}_i(C, B_i)$. These tables are actually much smaller since they have fewer input bits. Overall, this may result in much smaller area, and symmetry still applies for each of the m TO_i . However, this table size reduction comes at the cost of $m - 1$ additions.

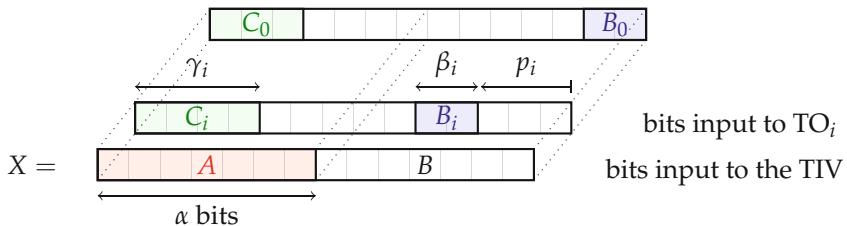


Fig. 17.9 Multipartite input word decomposition.

Let us define $p_0 = 0$, and $p_i = \sum_{j=0}^{i-1} \beta_j$ for $i > 0$. The function computed by the TO is then:

$$\begin{aligned}
 \text{TO}(C, B) &= s(C) \times \sum_{i=0}^{m-1} 2^{p_i} B_i \\
 &= \sum_{i=0}^{m-1} s(C) \times 2^{p_i} B_i \\
 &= \sum_{i=0}^{m-1} \text{TO}_i(C, B_i).
 \end{aligned} \tag{17.2}$$

This improvement thus entails two trade-offs:

- A cost trade-off, between the cost of the increased number of additions and the reduction of table size.
- An accuracy trade-off: Eq. (17.2) is not an approximation, but it will lead to more discretization errors (one per table) which will sum up to a larger global discretization error, unless the smaller tables have a bigger output accuracy (and thus are bigger). This trade-off will be formalized in a parametric error analysis in the sequel.

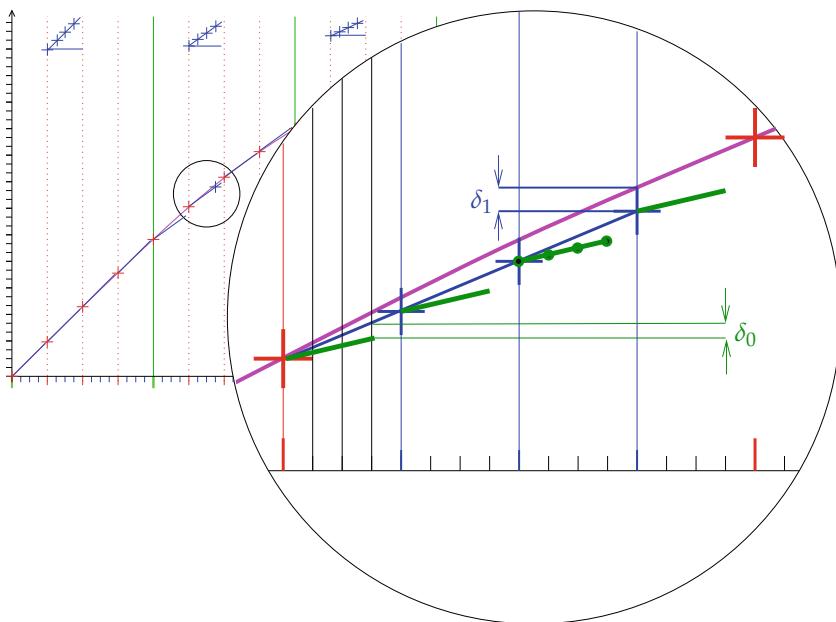


Fig. 17.10 A zoom view on the multipartite approximation ($m = 2$, without symmetry).

This decomposition can be improved further. Remark, in Eq. (17.2), that for $j > i$, there is a factor $2^{p_j - p_i}$ between the formula giving TO_j and that of TO_i . In other terms, TO_i is more accurate than TO_j . It will be possible,

therefore, to build even smaller tables than Schulte and Stine by compensating the (wasted) higher accuracy of TO_i by a rougher approximation on $s(C)$, obtained by removing some least significant bits from the input C . Figure 17.10 illustrates this in the case when $m = 2$: There, the slope of the smaller green segments is a worst approximation to the slope of the function curve. However, the corresponding error δ_0 is smaller than the error δ_1 of the larger blue segment. Muller's approach [Mul99] exploited this idea in a specific case of regular input word decomposition, with an error analysis based on Taylor formula.

All the previous approaches can be unified [DT05] by considering arbitrary decompositions of the input word that include Schulte and Stine's, Muller's, but also many others. The algorithm is then simply to enumerate all these possible decompositions. The multipartite approximation error is a function of the decomposition: Some decompositions do not allow for faithful rounding and are rejected. Among the remaining ones, the decomposition allowing for the cheaper architecture is selected.

The generalized multipartite decomposition is described in Fig. 17.9:

- The input word is split into two sub-words A and B of respective sizes α and β with $\alpha + \beta = w_X$.
- The most significant sub-word A addresses the TIV.
- The least significant sub-word B will be used to address $m \geq 1$ TOs.
 - B will in turn be decomposed into m sub-words B_0, \dots, B_{m-1} , the least significant being B_0 .
 - A sub-word B_i starts at position p_i and consists of β_i bits. We have $p_0 = 0$ and $p_{i+1} = p_i + \beta_i$.
 - The sub-word B_i is used to address the TO_i , along with a sub-word C_i of length γ_i of A .
 - The TO_i will compute a linear function

$$\text{TO}_i(C_i, B_i) \approx s(C_i) \times 2^{p_i} B_i. \quad (17.3)$$

- To simplify notations, we will denote

$$\mathcal{D} = \{\alpha, \beta, m, (\gamma_i, p_i, \beta_i)_{i=0 \dots m-1}\}$$

such a decomposition.

The original bipartite decomposition is a special case of multipartite decomposition with $m = 1$, $\alpha \approx 2w_X/3$, $\gamma_0 \approx w_X/3$, $p_0 = 0$, $\beta = \beta_0 \approx w_X/3$. Schulte and Stine's method [SS99b] is a multipartite decomposition where all the C_i 's are equal, while Muller's multipartite approach [Mul99] is a decomposition where all the γ_i are multiples of some integer. However, the optimal decomposition is usually none of these [DT05].

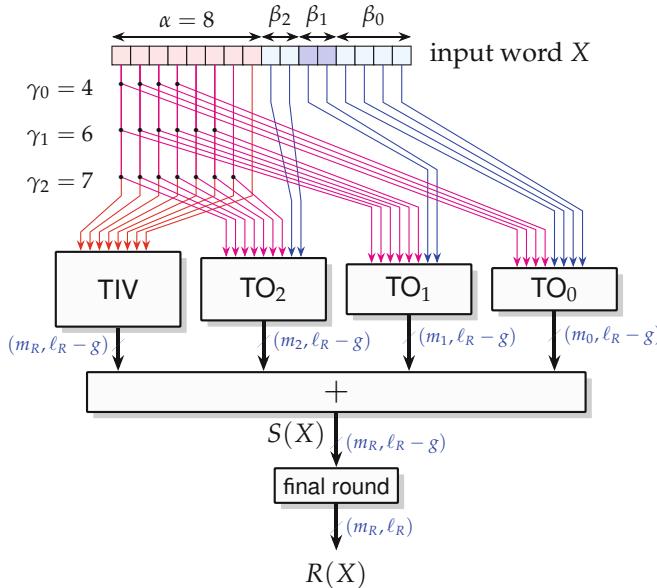


Fig. 17.11 Example multipartite architecture.

Such a decomposition corresponds to a parametric architecture, described in Fig. 17.11. As this figure shows, the implementation of each TO_i can still exploit their symmetry.

So far we have exposed a parameterized architecture with a lot of parameters. We thus follow the methodology presented in Chap. 3. It remains to write an error analysis that includes approximation and rounding errors and then implement it as part of a program that chooses the “best” value of each parameter.

17.2.5 Error Analysis

The objective of this section and the following is to determine the best multipartite architecture (in terms of speed or area) that allows for a faithful (or last-bit accurate) operator, as defined in Chap. 3.

The total absolute error of the architecture is defined as the difference between the computed value and the function:

$$\delta_{\text{total}}(X) = R(X) - f(X). \quad (17.4)$$

Last-bit accuracy means that this error should be smaller than the ulp of the output:

$$|\delta_{\text{total}}| < 2^{\ell_R}. \quad (17.5)$$

This error is the sum of approximation errors and rounding errors. To capture this distinction, we introduce a virtual value, the multipartite approximation function for a decomposition \mathcal{D} , denoted $\tilde{f}^{\mathcal{D}}(X)$. It is simply the piecewise linear function plotted in Figs. 17.6 and 17.10, for instance. This function is formally defined as

$$\tilde{f}^{\mathcal{D}}(X) = \text{TIV}^*(A) + \sum_{i=0}^{m-1} \text{TO}_i^*(C_i, B_i) \quad (17.6)$$

where A and B_i are the sub-words of the input described in Fig. 17.9, $\text{TIV}^*(A)$ is a real-valued initial value, and $\text{TO}_i^*(C_i, B_i)$ is a real-valued table of offsets.

Equation (17.4) becomes

$$\delta_{\text{total}}(X) = \underbrace{R(X) - \tilde{f}^{\mathcal{D}}(X)}_{=\delta_{\text{round}}(X)} + \underbrace{\tilde{f}^{\mathcal{D}}(X) - f(X)}_{=\delta_{\text{approx}}(X)}. \quad (17.7)$$

Let us first focus on the approximation error

$$\delta_{\text{approx}}(X) = \tilde{f}^{\mathcal{D}}(X) - f(X) \quad (17.8)$$

$$= \text{TIV}^*(A) + \sum_{i=0}^{m-1} \text{TO}_i^*(C_i, B_i) - f(X). \quad (17.9)$$

Our objective is to find the values of $\text{TIV}^*(A)$ and $\text{TO}_i^*(C_i, B_i)$ (all real numbers) that minimize $\delta_{\text{approx}}(X)$ for a given decomposition \mathcal{D} .

It will help that each TO_i^* is really only determined by its slopes $s(C_i)$ for each value of C_i (see Figs. 17.10 and 17.8), and each slope may be determined independently of the others.

For this purpose, let us assume that f is a monotonic function with monotonic derivative (i. e., convex or concave) on its domain. This is not a very restrictive assumption: It is the case, after argument reduction, of all the functions studied by previous authors.

Figure 17.12 (another zoom on Fig. 17.6) depicts the approximation due to one table in this case. This figure assumes an ideal initial value for each segment (the actual initial value will be the sum of $\text{TIV}^*(A)$ and the contributions of other tables).

As the figure suggests, with our hypothesis of a monotonic (decreasing in the figure) derivative, the approximation error is maximal on the borders of the interval on which the segment slope is constant. The minimum $\delta_i(C_i)$ of this maximum error is obtained when

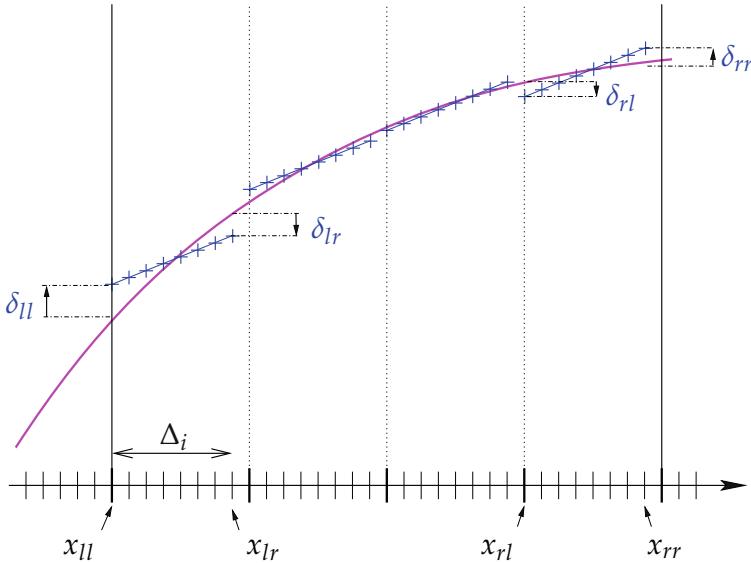


Fig. 17.12 Computing the approximation error of one multipartite table.

$$\delta_{ll} = -\delta_{lr} = -\delta_{rl} = \delta_{rr} = \delta_i(C_i) \quad (17.10)$$

with the notations of Fig. 17.12. This system of equations is easily expressed in terms of $s(C_i)$, p_i , β_i , γ_i , f and the initial values on the extremal intervals. Solving this system gives the optimal slope and the corresponding error:

$$s_i(C_i) = \frac{f(x_{lr}) - f(x_{ll}) + f(x_{rr}) - f(x_{rl})}{2\Delta_i} \quad (17.11)$$

$$\delta_i(C_i) = \frac{f(x_{lr}) - f(x_{ll}) + f(x_{rr}) - f(x_{rl})}{4} \quad (17.12)$$

where

$$\Delta_i = 2^{l_x+p_i}(2^{\beta_i} - 1), \quad (17.13)$$

$$x_{ll} = 2^{-\gamma_i}C_i, \quad (17.14)$$

$$x_{lr} = x_{ll} + \Delta_i, \quad (17.15)$$

$$x_{rl} = x_{ll} + 2^{-\gamma_i} - 2^{l_x+p_i+\beta_i}, \quad (17.16)$$

$$x_{rr} = x_{rl} + \Delta_i. \quad (17.17)$$

Remark that the slope that minimizes the error (17.11) is the average value of the slopes on the borders of the interval. This is not surprising. Some previous works used the slope at the midpoint of this interval, which is a good approximation for smooth functions.

Now, the error $\delta_i(C_i)$ depends on C_i , in other words, on the interval on which the slope is considered constant. Using the same argument of convexity, it will be maximum either for $C_i = 0$ or for $C_i = 2^{\gamma_i} - 1$. Finally, the maximum approximation error due to TO_i in the decomposition \mathcal{D} is

$$\overline{\delta_i^{\mathcal{D}}} = \max(|\delta_i(0)|, |\delta_i(2^{\gamma_i} - 1)|). \quad (17.18)$$

A bound on the overall multipartite approximation error can then be computed as

$$\overline{\delta_{\text{approx}}} = \sum_{i=0}^{m-1} \overline{\delta_i^{\mathcal{D}}}. \quad (17.19)$$

In practice, it is easy to compute this approximation error by implementing Eqs. (17.12) to (17.19). Altogether it requires a few floating-point operations per TO_i .

17.2.5.1 Optimal TIV and TO Values

An initial value $\text{TIV}^*(A)$ provided by the TIV for an input sub-word A – here interpreted as an integer in $[0, 2^\alpha - 1]$ – will be used on an interval $[x_l(A), x_r(A)]$ defined (using the previous notations) by

$$x_l(A) = 2^{-\alpha} A, \quad (17.20)$$

$$x_r(A) = x_l(A) + \sum_{i=0}^{m-1} \Delta_i. \quad (17.21)$$

On this interval, each TO_i provides a constant slope, as its C_i is a sub-word of A . The approximation error, which is the sum of the $\delta_i(C_i)$ defined by Eq. (17.12), will be maximal (with opposite signs) for x_l and x_r .

The TIV exact value that ensures that this error bound is reached is therefore (before rounding)

$$\text{TIV}^*(A) = \frac{f(x_l) + f(x_r)}{2} \quad (17.22)$$

and the TO_i values before rounding are (see Fig. 17.8)

$$\text{TO}_i^*(C_i, B_i) = s(C_i) \times 2^{\ell_R + p_i} \left(B_i + \frac{1}{2} \right). \quad (17.23)$$

17.2.5.2 Rounding Errors

It remains to actually fill the tables. Unfortunately, we cannot simply round the above ideal values to the target precision ℓ_R . Each table would entail a maximum rounding error of 2^{ℓ_R-1} , meaning that the total error budget of 2^{ℓ_R} is unachievable as soon as there are two tables plus some approximation error. Each table will therefore be filled with a precision that is g bits better than the target precision (g stands for “guard bits”).

From (17.7), the rounding error is defined as

$$\delta_{\text{round}}(X) = R(X) - \widetilde{f}^D(X) \quad (17.24)$$

$$= \underbrace{R(X) - S(X)}_{=\delta_{\text{final round}}(X)} + \underbrace{S(X) - \widetilde{f}^D(X)}_{=\delta_{\text{rnd_all_tables}}(X)} \quad (17.25)$$

where $S(X)$ is the extended-precision sum computed by the architecture (see Fig. 17.11), and

$$\delta_{\text{rnd_all_tables}}(X) = S(X) - \widetilde{f}^D(X) \quad (17.26)$$

$$\begin{aligned} &= \text{TIV}(A) + \sum_{i=0}^{m-1} \text{TO}_i(C_i, B_i) \\ &\quad - \text{TIV}^*(A) + \sum_{i=0}^{m-1} \text{TO}_i^*(C_i, B_i) \end{aligned} \quad (17.27)$$

This describes $m+1$ errors that correspond to filling the $m+1$ tables: $\text{TIV}(A) - \text{TIV}^*(A)$ when filling the TIV and $\text{TO}_i(C_i, B_i) - \text{TO}_i^*(C_i, B_i)$ when filling the m TO_i . With rounding to nearest, we have for each table the same bound to this rounding error:

$$|\text{TIV}(A) - \text{TIV}^*(A)| \leq 2^{\ell_R-g-1} \quad (17.28)$$

$$\forall i \quad |\text{TO}_i(C_i, B_i) - \text{TO}_i^*(C_i, B_i)| \leq 2^{\ell_R-g-1} \quad (17.29)$$

and the bound 2^{ℓ_R-g-1} can be made as small as desired by increasing g .

The sum of all these rounding errors can therefore be bounded by

$$|\delta_{\text{rnd_all_tables}}| \leq (m+1)2^{\ell_R-g-1} \quad (17.30)$$

and again, this bound can be made as small as desired by increasing g .

However, the final summation is now also performed on g more bits than the target precision. The final sum $S(X)$ must be rounded to the target precision, entailing a new rounding error $\delta_{\text{final round}}$ bounded by

$$|\delta_{\text{final round}}| \leq 2^{\ell_R-1}.$$

The HUB trick [DM95] may improve it to

$$|\delta_{\text{final round}}| \leq 2^{\ell_R - 1} (1 - 2^{-g}). \quad (17.31)$$

This trick (which also ensures the $2^{-\ell_R - g - 1}$ error bound for the TO_i in the presence of symmetry) will be presented in Sect. 17.2.7.

17.2.5.3 Computing the Number of Guard Bits

The condition (17.5) to ensure faithful rounding is rewritten $\bar{\delta}_{\text{final round}} + \bar{\delta}_{\text{rnd_all_tables}} + \bar{\delta}_{\text{approx}} < 2^{\ell_R}$. Taking the bounds on $\bar{\delta}_{\text{final round}}$ and $\bar{\delta}_{\text{rnd_all_tables}}$ from (17.31) and (17.30), respectively, this condition becomes

$$g > \ell_R - 1 + \log_2 m - \log_2 (2^{\ell_R - 1} - \bar{\delta}_{\text{approx}}). \quad (17.32)$$

If $\bar{\delta}_{\text{approx}} \geq 2^{\ell_R - 1}$, the decomposition \mathcal{D} is unable to provide the required output accuracy. Otherwise the previous inequality gives us the minimal number g of extra bits that ensures faithful rounding:

$$g = \lceil \ell_R - 1 + \log_2 m - \log_2 (2^{\ell_R - 1} - \bar{\delta}_{\text{approx}}) \rceil. \quad (17.33)$$

Our experiments show that in practice, it is very often possible to decrease this value by one and still keep faithful rounding. This is due to the actual worst-case rounding error in each table being smaller than the one assumed above, thanks to the small number of entries for each table. In the FloPoCo implementation we choose to keep the value given by (17.33).

17.2.6 Computing the Sizes of the TO_i

The TO_i consist of $2^{\gamma_i + \beta_i}$ entries, and we now need to compute the size of each of these entries. The TO_i have a smaller range than the TIV. The range of $\text{TO}_i^*(C_i, *)$ without symmetry is exactly equal to $|s_i(C_i) \times \Delta_i|$ (see Fig. 17.12). Again, for convexity reasons, this range is maximum either on $C_i = 0$ or $C_i = 2^{\gamma_i} - 1$:

$$r_i = \max(|s_i(0) \times \Delta_i|, |s_i(2^{\gamma_i} - 1) \times \Delta_i|). \quad (17.34)$$

The MSB of the TO_i is therefore

$$m_i = \lceil \log_2 r_i \rceil \quad (17.35)$$

Without symmetry, the size in bits of the TO_i is $2^{\gamma_i + \beta_i}(m_i - \ell_R + g + 1)$.

In a symmetrical implementation of the TO_i , the size in bits of the corresponding table will be $2^{\gamma_i + \beta_i - 1}(m_i - \ell_R + g)$. Symmetry wins one bit on input and also one bit on output (see Fig. 17.8).

17.2.7 Filling the Tables Using HUB Format

HUB rounding was used (although not under this name) by Das Sarma and Matula in their first bipartite paper [DM95]. Stine and Schulte generalized its use to multipartite tables in [SS99b].

First, following Sect. 3.1.6 of Chap. 3, let us remind the reader that there are two ways to round a real number to $\ell_R - g$ bits with an error smaller than $2^{\ell_R - g - 1}$:

- The most natural way is round to the nearest.
- Another method is to use an intermediate half-unit biased (HUB) format, for the ulp value $2^{\ell_R - g}$. In this case the number is truncated to precision $\ell_R - g$, and an implicit 1 is assumed at position $\ell_R - g - 1$.

Of course, if we use HUB rounding for some of the tables, the corresponding implicit half-ulps must be added explicitly at some point. Here we remark that any *even* number of half-ulps becomes a multiple of the ulp and can therefore be added, for free, to all the values of one of the tables, for instance, the TIV.

However, it is also desirable to obtain the sum (before the final rounding) in HUB format, because it reduces the error of the final rounding from $\bar{\delta}_{\text{rnd_final}} = 2^{\ell_R - 1}$ to $\bar{\delta}_{\text{rnd_final}} = 2^{\ell_R - 1} - 2^{\ell_R - g - 1}$ – this was illustrated by Table 3.2 in Sect. 3.1.6.

Here, we have $m + 1$ tables. HUB rounding also simplifies exploitation of symmetry (we remind that the opposite of an HUB number is obtained by simple complementation). Therefore, we choose to use HUB rounding for all the m TO_i .

Concerning the TIV, the good choice of rounding depends on the parity of m :

- If m is even, then we use HUB rounding for it as well. The m implicit half-ulps of the TO_i are explicated by adding $m/2$ ulps to each TIV entry. Only the implicit half-ulp due to HUB rounding the TIV remains in the sum.
- If m is odd, then we use round to the nearest for the TIV. We explicit $m - 1$ half-ulps by adding the constant $(m - 1)/2$ ulps to each TIV entry. One implicit half-ulp remains in the sum.

Let us now address the final rounding. After summing the TIV and the TO_i , we need to round the sum to ℓ_R , the precision of the output. This also

can be done by simply truncating the sum (at no hardware cost), provided we have added $2^{\ell_R - 1}$ to each value of the TIV when filling it.

Summing it up, the integer values that should fill the TOis are

$$\text{TO}_i(C_i B_i) = \left\lfloor 2^{-\ell_R} \text{TO}_i^*(C_i B_i) \right\rfloor \quad (17.36)$$

and the values that should fill the TIV are, if m is odd,

$$\text{TIV}(A) = \left\lfloor 2^{-\ell_R} \text{TIV}^*(A) + \frac{m-1}{2} + 2^{g-1} \right\rfloor \quad (17.37)$$

and if m is even:

$$\text{TIV}(A) = \left\lfloor 2^{-\ell_R} \text{TIV}^*(A) + \frac{m}{2} + 2^{g-1} \right\rfloor. \quad (17.38)$$

17.2.8 Errorless Compression of the TIV

The size in bits of the TIV as represented in Fig. 17.11 is $2^\alpha(w_R + g)$ where $w_R = m_R - \ell_R + 1$ is the output width.

A final improvement to the multipartite method was the observation by Hsiao that the TIV could be further compressed [Hsi+15]. The method for this errorless compression has been reviewed in Sect. 17.1. As this compression is errorless, it does not impact the error analysis.

Figure 17.13 shows an example multipartite architecture with compressed TIV, along with the fixed-point alignment of the table output. The final rounding block has been removed from this figure since it is implemented for free by truncation of the sum, as per the previous section.

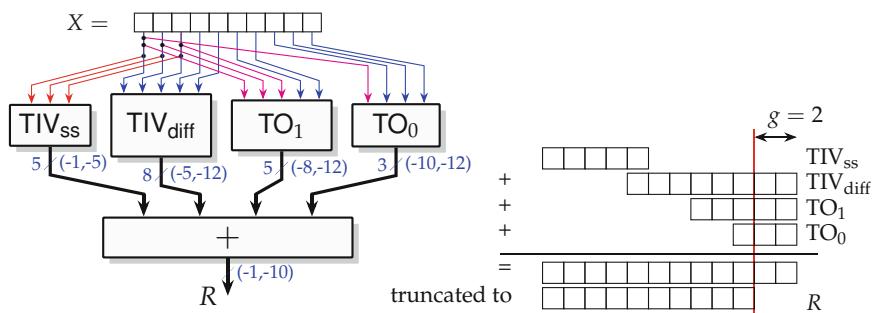


Fig. 17.13 Example multipartite architecture with TIV compression.

17.2.9 Putting It All Together: The Multipartite Optimization Algorithm

Algorithm 17.3 describes an efficient architecture exploration algorithm based on the previous error analysis.

Algorithm 17.3: Enumerating the multipartite decompositions.

1. Enumerate the decompositions $\mathcal{D} = \{\alpha, \beta, m, (\gamma_i, p_i, \beta_i)_{i=0..m-1}\}$.
 2. For each decomposition \mathcal{D} :
 - a. Compute the approximation error bound $\bar{\delta}_{\text{approx}}$ by (17.19). This error depends on the decomposition, but not on g . Keep only those decompositions for which $\bar{\delta}_{\text{approx}} < 2^{\ell_R - 1}$.
 - b. Compute g out of (17.33). This value of g ensures a faithful architecture.
 - c. Knowing g , compute the total size of all the tables for an architecture implementing \mathcal{D} , as per Sect. 17.2.6. The TIV compression optimization is also performed in this step.
 - d. If this size is smaller than the current best, keep this \mathcal{D} as the best candidate decomposition.
-

Among the parameters of a decomposition, m (the number of tables) has a special status. Indeed, it can be viewed as a user-controlled knob that controls the trade-off between tables and additions. In FloPoCo, the default interface enumerates all the possible values of m (which ranges from 1 to 7 at most for practical sizes), but it is also possible to provide the value of m at the command line.

For a given m , all the other parameters must be enumerated, as the best decomposition really depends on the numerical properties of the function. Enumerating all the possible decompositions is an exponential task, but the design space can be reduced as follows, cutting the exploration time down to a few seconds for 24-bit operands (the maximum size for which multipartite methods architectures make sense):

- For a given pair (p_i, β_i) , the error bound $\bar{\delta}_i^{\mathcal{D}}$ grows as γ_i decreases. There exists a γ_{\min} such that for any $\gamma_i \leq \gamma_{\min}$, this error is larger than $2^{\ell_R - 1}$. These $\gamma_{\min}(p_i, \beta_i)$ may be computed once and stored in a table.
- The enumeration of the (p_i, β_i) is limited by the relation $p_{i+1} = p_i + \beta_i$, and the enumeration on γ_i is limited by $\gamma_{\min} < \gamma_i < \alpha$.

Another observation that speeds up the exploration is the following. For a given function f , input size ℓ_X , and number of TOs m , the bound $\bar{\delta}_i^{\mathcal{D}}$ to the

approximation error due to a TO_i is only a function of the three parameters p_i , β_i , and γ_i of this TO_i . It is therefore possible to precompute all the possible errors δ_i^D and store them in a three-dimensional array. The size of this array is at most $24^3 = 13,824$ double-precision floating-point numbers.

This methodology is implemented in FloPoCo (an older implementation in Java can be found on the Internet). As a final check, the implementation computes the overall error out of the actual tables and checks that it is indeed within the one ulp bound. Figure 17.14 is an example plot of this error.

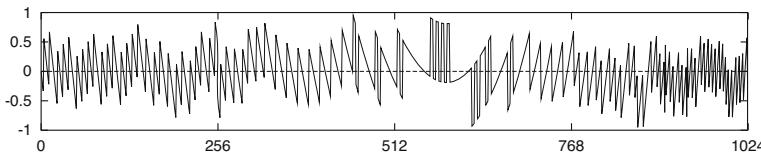


Fig. 17.14 Plot of the evaluated overall error $\delta_{\text{total}}(X)$ for a 10-bit sine when $m = 2$.

Hands on: Observing the multipartite exploration algorithm

The following command builds a 16-bit in, 16-bit out ufix($-1, -16$) sine approximation.

```
flopoco FixFunctionByMultipartiteTable \
    f="(1b16-1)/1b16*sin(pi/2*x)" signedIn=false \
    lsbIn=-16 lsbOut=-16
```

In this case the exploration takes less than one second. The reader is invited to have a look at the messages output on the console: Values of m attempted range from 1 to 3; there are 5 valid decompositions with $m = 1$, and there are 39 with $m = 2$ and 24 with $m = 3$. The value of g ranges from 1 to 5.

The best option is reported as $m = 3$ with rather balanced tables: Their sizes are $7 \cdot 2^6 + 13 \cdot 2^9$ bits for the compressed TIV and $9 \cdot 2^9 + 7 \cdot 2^8 + 5 \cdot 2^8$ bits for the 3 TO, for a total of 14784 bits. A plain table would store $16 \cdot 2^{16} = 1,048,576$ bits (1 Mbit): We have a table compression by a factor 70 here.

Optionally, it is possible to force a value for m in parameter nbTO and to disable TIV compression.

The results obtained by the automatic method presented above can often be slightly improved, up to 15% in terms of table sizes [DT05]. The reason is that the automatic algorithm assumes that worst-case rounding will be attained in filling the tables, which is not the case. As we have many TO_i with few entries (e.g., typically 2^5 to 2^9 entries for 16-bit operands), there is statistically a good chance that the sum of the table-filling rounding errors is

significantly smaller than predicted. This is a random effect which can only be tested by an exhaustive check of the architecture. However, in a final stage, it may be worth trying several slight variations of the parameters, which can be of two types:

- g may be decremented.
- Some of the γ_i can be decremented (meaning less accurate slope).

Table 17.2 gives the example of such a lucky case, with 11.5% improvement in size. Here, the values of the γ_i even produce a method error of more than 0.5 ulp, which the lucky rounding compensates.

Table 17.2 Effect of manual optimization of the parameters for $\sin(\pi/4 \cdot x)$, $\ell_X = \ell_R = -16$, $m = 4$.

γ_i	g	Size	Max. measured error in ulp
Automatic	6 6 5 5 3 7808		0.915
Hand-tuned	6 5 4 3 3 6912		0.939

17.2.10 The Issue of Non-Monotoncities

The objective of faithful rounding unfortunately allows for non-monotoncities in the implementation of a monotonic function. This is illustrated by Fig. 17.15, where the reader may check that the function is never more than one ulp away from the curve. These non-monotoncities are never bigger than one ulp, and it is a problem of all the faithful approximation schemes, but the multipartite method as presented makes it happen quite often, as Fig. 17.12 (around $x = x_{rl}$) illustrates.

The subject of monotonicity in the context of bipartite tables has been studied by Iordache and Matula [IM99]. They reverse-engineered the AMD K6-2 implementation of fast reciprocal and reciprocal square root instructions, part of the 3D-Now instruction set extensions. They found that bipartite approximations were used, that the reciprocal was monotonic, and that the reciprocal square root was not. They also showed that the latter could be tuned to become monotonic, at the expense of a larger table size (7.25 KB instead of 5.5 KB). This tuning involves increasing the output size of the TIV, and an exhaustive exploration of what value these extra bits should take.

In general, if monotonicity is an important property, it can be enforced simply in a multipartite approximation, by using appropriate slopes and TIV. For instance, monotonically increasing functions with decreasing derivatives (as in our figures) may use the slope on the right of the interval instead

of the middle, ensuring that the approximation is monotonic. This means a larger maximum approximation error, however. Rounding errors can then be kept within bounds that ensure monotonicity by increasing g as in [IM99]. All this will entail increased hardware cost. To our knowledge, a general and systematic study of this question remains to be done.

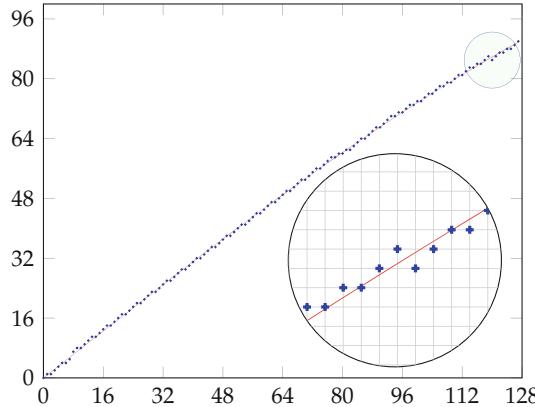


Fig. 17.15 Non-monotonicity in a faithful approximation of $\sin(\frac{\pi}{4}x)$.

17.2.11 Toward ILP-Optimized Multipartite Architectures

It is possible [DD22] to formulate the multipartite optimization problem as an ILP one, using an approach that has been used for other function approximation techniques [De +17]. The idea is to have as many ILP variables as there are table entries and to generate, for each possible input value, two constraints that capture the faithful or correct rounding property. Obviously the size of the ILP scales exponentially with w_X , and for this reason the approach is currently limited to less than 16 bits. In addition, some ILP engineering is needed to express the cost function, and this is still work in progress at the time of writing this book.

This approach provides slightly smaller architectures than the previous one for faithfully rounded functions. Its main advantage, however, is that it allows for correctly rounded implementations, as it does not need to bound separately approximation and rounding errors.

A correctly rounded multipartite method can be viewed as another lossless table compression technique. In preliminary experiments on 16-bit functions, with all the tables implemented as LUTs on FPGAs, the LUT count may be reduced by a factor 5 compared to LDTC compression [DD22]. This comes at the cost of a larger compressor tree, but interestingly, the latency of

the multipartite-compressed table is also lower than that of a plain table, so this is a win-win compression. This is probably due to the fact that a more compact solution entails less routing delay.

17.2.12 Conclusion: Multipartite Architectures Are Close to Optimal Among Order-One Methods

Consider the family of linear (order-one) approximation schemes using some α bits of the input to address a TIV. There is an intrinsic lower bound on α for this family, and it is the α for which the (mathematical) approximation error prevents faithful rounding. Generally speaking, this bound is about $w_X/2$ as given by the Taylor formula (and $w_X/3$ for order-two methods, etc.). This bound can be computed for each function exactly.

We have observed empirically the best multipartite decomposition is almost always exhibiting the smallest α compatible with a faithful approximation.

Combined with the observation that the main contribution to the total size is always the TIV, this allows us to claim that the best multipartite approximations are close to the global optimal in the family of linear approximation schemes. More accurately, even the availability of a free and perfect multiplier to implement a linear scheme will remove only the TO_i , which account for less than half the total size.

17.3 Other Table-and-Addition Methods

This section reviews, more quickly, other table-and-addition methods that have been found less efficient in practice than the previous generalized multipartite method.

17.3.1 Addition-Table-Addition Methods

The addition-table-addition (ATA) methods, introduced by Wong and Goto [WG95], allow additions before and after the table lookups. A whole range of such methods is possible.

Let us again consider the input word decomposition of Fig. 17.4, which we note

$$X = A + 2^{-\beta}B$$

with the additional condition that $\alpha > \beta$.

To compute $f(A + 2^{-\beta}B)$, it is possible to use the first-order Taylor approximation

$$f(A + 2^{-\beta}B) \approx f(A) + 2^{-\beta}Bf'(A)$$

The core of the ATA methods is that $Bf'(A)$ can be approximated as follows:

$$Bf'(A) \approx f(A + B) - f(A).$$

Finally,

$$f(A + 2^{-\beta}B) \approx f(A) + 2^{-\beta}(f(A + B) - f(A)).$$

In other terms, this first-order ATA method approximates, in a neighborhood of A , the graph of f with an homothetic reduction of this graph with a ratio of 2^β , as pictured in Fig. 17.16.

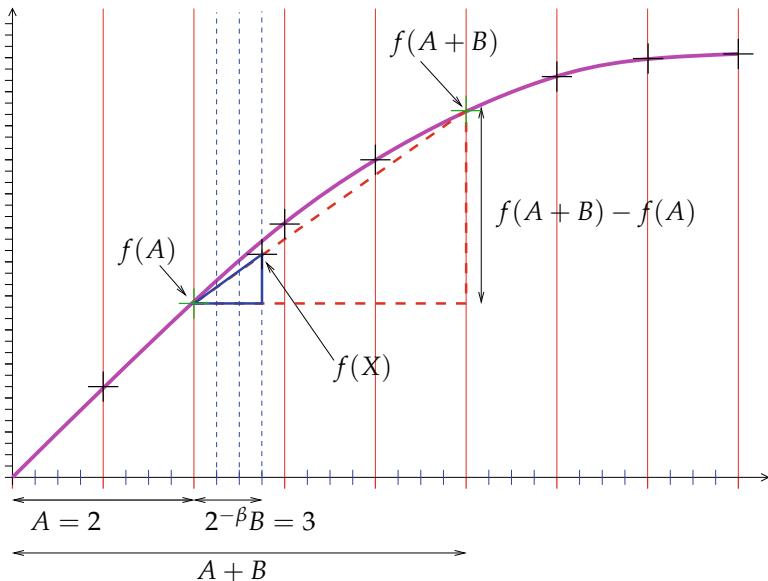


Fig. 17.16 First-order ATA.

Evaluating $f(x)$ thus involves:

- one α -bit addition to compute $A + B$,
- two lookups in the same table, retrieving $f(A)$ and $f(A + B)$,
- one subtraction to compute the difference between the previous two lookups. The size of this subtraction (less than α bits) depends on f and β and may be computed exactly,
- one constant shift by β bits to perform the division by 2^β ,
- one final addition, on w_R bits.

Both table lookups can be performed in parallel in a dual-port table, or in parallel using two tables, or in sequence/pipeline (one read before the addition and one read after). This leads to a range of architectural trade-offs.

This method can be extended to the second order by using a *central difference formula* to compute a much better approximation of the derivative (the error is a third-order term) as depicted in Fig. 17.17.

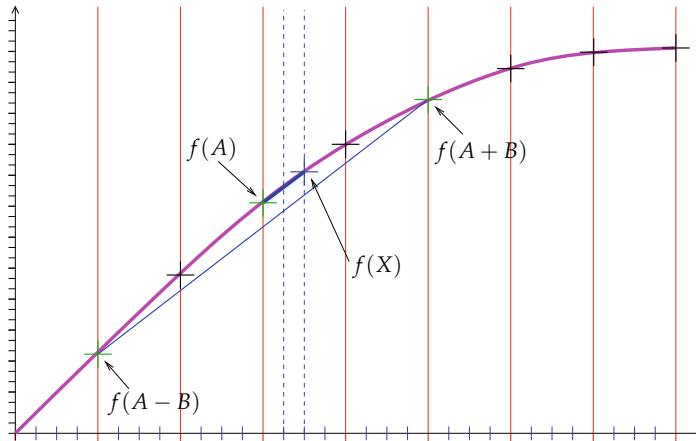


Fig. 17.17 Second-order ATA.

The formula used is now

$$Bf'(A) \approx \frac{f(A+B) - f(A-B)}{2}$$

and the algorithm consists of the following steps:

- Compute (in parallel) $A + B$ and $A - B$.
- Read in a table $f(A)$, $f(A + B)$, and $f(A - B)$.
- Compute

$$f(A + 2^{-\beta}B) \approx f(A) + 2^{-\beta-1}(f(A+B) - f(A-B)).$$

We now need three lookups in the same table and 7 additions. Here again, a range of space/time trade-offs is possible.

The original method by Wong and Goto [WG95] is actually more sophisticated: As in the STAM method, they split B into two sub-words of equal sizes $B = B_1 + 2^{\beta/2}B_2$ and distribute $Bf'(A)$ using two centered differences, which reduces table sizes. Besides they add a table which contains second- and third-order corrections, indexed by the most significant half-word of A

and the most significant half-word of B . For 24 bits of precision, their architecture therefore consists of 6 tables with 12 or 13 bits of inputs and a total of 9 additions.

Another option would be to remark that with these three table lookups, it is also possible to use a 2nd-order Taylor formula:

$$f(A + 2^{-\beta}B) \approx f(A) + 2^{-\beta}Bf'(A) + \frac{(2^{-\beta}B)^2}{2}f''(A)$$

Indeed we may approximate the term $f''(A)$ by

$$\begin{aligned} f''(A) &\approx \frac{f'(A + B/2) - f'(A - B/2)}{B} \\ &\approx \frac{\frac{f(A+B)-f(A)}{B} - \frac{f(A)-f(A-B)}{B}}{B} \\ &\approx \frac{f(A+B) - 2f(A) + f(A-B)}{B^2}. \end{aligned}$$

And finally,

$$\begin{aligned} f(A + 2^{-\beta}B) &\approx f(A) + 2^{-\beta-1} (f(A+B) - f(A-B)) \\ &\quad + 2^{-2\beta-1} (f(A+B) - 2f(A) + f(A-B)). \end{aligned}$$

However, the larger number of lookups and arithmetic operations entails more rounding errors which actually consume the extra accuracy obtained thanks to this formula.

Finally, the ATA methods can be improved using symmetry, just like multipartite methods.

These methods have been found [DT05] to perform better than the original bipartite approaches, but worse than the generalized multipartite. This is also true of the initial ATA architecture by Wong and Goto [WG95].

17.3.2 Partial Product Arrays

This method is due to Hassler and Tagaki [HT95]. First, the function is approximated by a polynomial of arbitrary degree (they use a Taylor approximation). Writing X and all the constant coefficients as sums of weighted bits (as in $X = \sum x_i 2^{-i}$), they distribute all the multiplications within the polynomial, thus rewriting the polynomial as the sum of a huge set of weighted products of some of the x_i .

A second approximation then consists in neglecting as many of these terms as possible, in order to be able to partition the remaining ones into several tables.

This idea is in principle very powerful, because the implementation space is very wide. However, for the same reason, it needs to rely on heuristics to explore this space. The heuristic chosen by Hassler and Tagaki in [HT95] leads to architectures which are less compact than their multipartite counterpart [SS99b] (and are interestingly similar). The reason is probably that the multipartite method exploits the higher-level property of continuity of the function, which is lost in the set of partial products.

The next chapter presents higher-order methods that do exploit such high-level properties.

References

- [BC07] Nicolas Brisebarre and Sylvain Chevillard. “Efficient Polynomial L^∞ - approximations”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2007, pp. 169–176 (cit. on p. 498).
- [CFD22] Maxime Christ, Luc Forget, and Florent de Dinechin. “Lossless Differential Table Compression for Hardware Function Evaluation”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 69.3 (2022), pp. 1642–1646 (cit. on pp. 497, 501, 502).
- [DD22] Orégane Desrentes and Florent de Dinechin. “Using integer linear programming for correctly rounded multipartite architectures”. In: *International Conference on Field-Programmable Technology (FPT)*. IEEE, 2022 (cit. on p. 522).
- [De +17] Davide De Caro, Ettore Napoli, Darjn Esposito, Gerardo Castellano, Nicola Petra, and Antonio GM Strollo. “Minimizing Coefficients Wordlength for Piecewise-Polynomial Hardware Function Evaluation With Exact or Faithful Rounding”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 64.5 (2017), pp. 1187–1200 (cit. on p. 522).
- [DM95] Debjit Das Sarma and David W. Matula. “Faithful Bipartite ROM Reciprocal Tables”. In: *12th Symposium on Computer Arithmetic*. Ed. by S. Knowles and W.H. McAllister. IEEE, 1995, pp. 17–28 (cit. on pp. 503, 507, 516, 517).
- [DT05] Florent de Dinechin and Arnaud Tisserand. “Multipartite Table Methods”. In: *IEEE Transactions on Computers* 54.3 (2005), pp. 319–330 (cit. on pp. 503, 510, 520, 526).
- [Hsi+15] Shen-Fu Hsiao, Po-Han Wu, Chia-Sheng Wen, and Pramod Kumar Meher. “Table Size Reduction Methods for Faithfully Rounded Lookup-Table-Based Multiplierless Function Evaluation”. In: *Transactions on Circuits and Systems II* 62.5 (2015), pp. 466–470 (cit. on pp. 497, 499, 503, 518).

- [Hsi+17] Shen-Fu Hsiao, Chia-Sheng Wen, Yi-Hau Chen, and KueiChun Huang. "Hierarchical Multipartite Function Evaluation". In: *Transactions on Computers* 66.1 (2017), pp. 89–99 (cit. on pp. 497, 499, 500, 503).
- [HT95] Hannes Hassler and Naofumi Takagi. "Function Evaluation by Table Look-up and Addition". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 1995, pp. 10–16 (cit. on pp. 526, 527).
- [IM99] Cristina Iordache and David W. Matula. "Analysis of Reciprocal and Square Root Reciprocal Instructions in the AMD K6-2 Implementation of 3DNow!" In: *Electronic Notes in Theoretical Computer Science* 24 (1999) (cit. on pp. 521, 522).
- [Mul99] Jean-Michel Muller. "A Few Results on Table-Based Methods". In: *Reliable Computing* 5.3 (1999), pp. 279–288 (cit. on pp. 503, 506, 508, 510).
- [SS99a] Michael J. Schulte and James E. Stine. "Approximating Elementary Functions with Symmetric Bipartite Tables". In: *IEEE Transactions on Computers* 48.8 (1999), pp. 842–847 (cit. on pp. 503, 507).
- [SS99b] James E. Stine and Michael J. Schulte. "The Symmetric Table Addition Method for Accurate Function Approximation". In: *Journal of VLSI Signal Processing* 21.2 (1999), pp. 167–177 (cit. on pp. 503, 506, 508, 510, 517, 527).
- [Sun+84] David A. Sunderland, Roger A. Strauch, Steven S. Wharfield, Henry T. Peterson, and Christopher R. Role. "CMOS/SOS Frequency Synthesizer LSI Circuit for Spread Spectrum Communications". In: *IEEE Journal of Solid-State Circuits* 19.4 (1984), pp. 497–506 (cit. on p. 503).
- [WG95] W. F. Wong and E. Goto. "Fast Evaluation of the Elementary Functions in Single Precision". In: *IEEE Transactions on Computers* 44.3 (1995), pp. 453–457 (cit. on pp. 523, 525, 526).

CHAPTER 18

Polynomial-Based Architectures for Function Evaluation

Polynomial approximation is a general technique that reduces the evaluation of some mathematical function to additions and multiplications only. This chapter studies polynomial approximation, as well as polynomial evaluation hardware computing just right. The addition of generic range reduction techniques brings in a trade-off between memory resources and computing resources.

The piecewise linear approximation of previous chapters was a particular case of polynomial approximation, with polynomials of degree 1. Polynomial approximation is a general technique for the evaluation of a numerical function f of one variable [Mul16]. It assumes that f is continuously differentiable over a closed interval I up to a certain order. In this chapter (as in the previous one), we will systematically normalize I to $[0, 1]$ or $[-1, 1]$, for the reasons given in Chap. 16.

This chapter refines several previous works [Lee+05; Lee+09; DJP10] describing generic function approximators: these are tools that input a function along with the parameters specifying its context and produce an architecture, as illustrated by Fig. 18.1.

The construction of a polynomial evaluator typically proceeds in three steps. The first is the computation of a polynomial that *approximates* the function f . The second is the generation of an architecture based on adders and

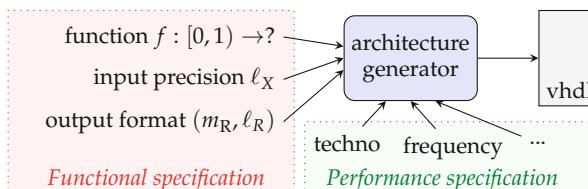


Fig. 18.1 Interface to a generic fixed-point function approximator.

Table 18.1 Common notations used in this chapter.

Symbol meaning
f A function
p A polynomial of one variable
i Integer, index of a monomial, as in $p(X) = \sum_{i=0}^d C_i X^i$
j Integer, index/position of a bit, as in $C_i = \sum_j 2^j c_{i,j}$
A Integer, index/address of a subinterval, address in a table
α Number of bits, size in bits of A

multipliers that will *evaluate* this polynomial. These two steps are usually preceded by a *range reduction* that transforms the initial function into one more suited for polynomial approximation [Mul16; Lee+09; Tho15]. These three steps are deeply linked by arithmetic considerations.

This chapter is organized as follows: Sect. 18.1 presents the state of the art in polynomial approximation. Section 18.2 presents the main generic techniques for range reduction and the trade-offs they enable. Section 18.3 discusses polynomial evaluation computing just right. Finally, Sect. 18.4 shows how to combine all these techniques into generators of polynomial approximators computing just right.

As a conclusion, we briefly review in Sect. 18.5 some works that use fixed-point polynomial approximators to implement floating-point functions.

Table 18.1 gives some common notations consistently used in this chapter, for future reference.

18.1 A Primer on Polynomial Approximation

Polynomial approximation is the generic mathematical tool that reduces the evaluation of a function to additions and multiplications. A good reference on polynomial approximation for function evaluation is Muller’s textbook [Mul16].

The polynomials in this chapter are polynomials of one real variable with real-valued coefficients.¹ A polynomial p of degree d of the real variable X is the function

$$p(X) = C_0 + C_1 X + C_2 X^2 + \dots + C_d X^d \quad \text{where } \forall i, C_i \in \mathbb{R}.$$

¹ The reader should be aware that this is considered boring in computer algebra circles, where your typical polynomial has many variables and is defined over an exotic composite field.

At some point, in order to build architectures, we will have to take into account that the polynomial coefficients C_i must be finitely representable as bit vectors. However, we start with the simpler case of arbitrary real coefficients.

Many techniques exist to approximate a function over an interval by a polynomial (Taylor polynomials, Chebyshev polynomials, Remez minimax polynomials among others) [Mul16]. When a polynomial p approximates a function f on an interval I , the quality of the approximation is defined by the error function

$$\delta_{\text{approx}}(X) = f(X) - p(X) \quad .$$

Let us take as a first example the approximation of the exponential function $f(X) = e^X$ on $[0, 1]$.

18.1.1 Taylor Approximation

A first idea is to use the Taylor formula: chose some $X_0 \in [0, 1]$, then

$$\begin{aligned} f(X) \approx f(X_0) + f'(X_0)(X - X_0) + \frac{f''(X_0)}{2!}(X - X_0)^2 + \dots \\ \dots + \frac{f^{(k)}(X_0)}{k!}(X - X_0)^k. \end{aligned} \tag{18.1}$$

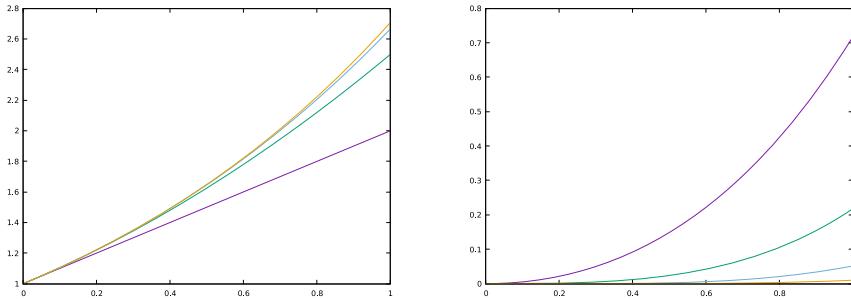
For the exponential function, the Taylor approximation polynomials of degrees 1 to 4 at point $X_0 = 0$ are

$$\begin{aligned} T_1(X) &= 1 + X \\ T_2(X) &= 1 + X + \frac{X^2}{2} \\ T_3(X) &= 1 + X + \frac{X^2}{2} + \frac{X^3}{6} \\ T_4(X) &= 1 + X + \frac{X^2}{2} + \frac{X^3}{6} + \frac{X^4}{24} \end{aligned}$$

Figure 18.2 plots these Taylor polynomials and the corresponding approximation error δ_{approx} on $[0, 1]$.

Hands on: Taylor Approximation in Sollya

Most computer algebra systems will offer you function approximation tools. We use here the open-source Sollya toolbox, which is a dependency of FloPoCo and therefore installed along with FloPoCo itself.

(a) Plots of T_1 (bottommost curve) to T_4 (topmost curve)(b) Plots of the approximation error for T_1 (topmost curve) to T_4 (bottommost curve)**Fig. 18.2** Degrees 1 to 4 Taylor approximation in 0 of $f(X) = e^X$, plotted on $[0, 1]$.

Sollya is a command-line interface: it can be launched by typing in a terminal the command `sollya`. A message is displayed, followed by a prompt. Below is the list of Sollya commands that were used to obtain the plots of Fig. 18.2. The reader is invited to try them out and then experiment with a different function, for instance, the function $f(X) = \sin(\frac{\pi}{2}X)$ studied in Chap. 16:

```
f=exp(x);
t1=taylor(f,1,0);
t2=taylor(f,2,0);
t3=taylor(f,3,0);
t4=taylor(f,4,0);
plot(t1, t2, t3, t4, [0;1]);
plot(f-t1, f-t2, f-t3, f-t4, [0;1]);
```

The resulting polynomials can be observed simply as follows:

```
t4;
```

Note that `f` and `t4` are not really functions, but rather expressions in which Sollya commands such as `taylor` look for a free variable (here `x`).

For more details on a Sollya command, use the `help` keyword, for instance:

```
help taylor;
```

Figure 18.2 first shows that the higher the degree, the more accurate the approximation. This is expected: it costs more (more additions and multiplications) to achieve better accuracy.

Figure 18.2 also shows that Taylor approximation is local: it is very accurate near X_0 (in Fig. 18.2 $X_0 = 0$) but becomes very inaccurate for values distant from 0. This can be solved by other analytical polynomial approx-

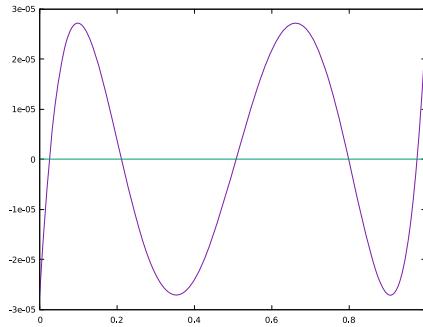


Fig. 18.3 Plot of $e^X - R_4(X)$ where R_4 is the degree-4 Remez approximation polynomial for e^X on $[0,1]$.

imation techniques, such as Chebyshev polynomials [Mul16]. However, in the following, we prefer to switch to numerical approximation techniques.

18.1.2 Remez Minimax Approximation

Remez devised in 1934 an algorithm [Rem34] that converges to a polynomial minimizing the maximum error over the whole interval. For this reason, this algorithm is also sometimes called *minimax* polynomial approximation, for instance, in the Maple software. The core of this algorithm is to make sure that the error function $\delta_{\text{approx}}(X)$ oscillates around zero, as illustrated by Fig. 18.3.

Table 18.2 compares the coefficients of a degree-4 Taylor approximation to those computed by the Remez algorithm. The reader may observe that the Remez coefficients remain close to the Taylor ones. In some situations, it may help to reason about Taylor coefficients to get a rough idea of the order of magnitude of the Remez coefficients.

Despite this, however, the worst-case approximation error of a Remez polynomial is typically much smaller than that of a Taylor polynomial of

Table 18.2 Comparison of Taylor and Remez coefficients for a degree-4 approximation of e^X on $[0, 1]$.

Coeff.	Taylor	Remez
C_0	1	1.000027162418764595123303558
C_1	1	0.998685400641021014694653206
C_2	0.5	0.510139460190883631832623651
C_3	0.1666...	0.139698148569719082021575408
C_4	0.041666...	0.069704494219892316564828091

the same degree, whatever the value of X_0 chosen for the latter. Table 18.3 shows how the two compared for a range of degrees on e^X on $[0, 1]$. This table will be very different for another function or on another interval, but the fact that Remez is more accurate than Taylor will remain true.

Table 18.3 Maximum approximation error versus degree for e^X on $[0, 1]$.

Degree	2	3	4	5
$\bar{\delta}_{\text{Taylor}}$	0.22 $(\approx 2^{-2})$	0.052 $(\approx 2^{-4})$	0.0099 $(\approx 2^{-6})$	0.0016 $(\approx 2^{-9})$
$\bar{\delta}_{\text{Remez}}$	0.0088 $(\approx 2^{-6})$	0.00054 $(\approx 2^{-10})$	0.000027 $(\approx 2^{-15})$	0.0000011 $(\approx 2^{-19})$

When designing hardware, the values of $\bar{\delta}_{\text{approx}}$ are not very expressive. It is more convenient to express them in bits by taking their logarithm in base two. Formally, an approximation polynomial can be used to design hardware that is last-bit accurate to an LSB no smaller than $\log_2(\bar{\delta}_{\text{approx}})$. Table 18.3 also provides this information. In the following, we will call $\log_2(\bar{\delta}_{\text{approx}})$ “accuracy in bits.”

Hands on: Remez Approximation in Sollya

Figure 18.3 was obtained thanks to the following Sollya commands:

```
f=exp(x) ;
p=remez(f, 4, [0;1]) ;
plot(f-p, [0;1]);
```

The reader is invited to change the degree (here 4) and observe how the error plot evolves.

The values of Table 18.3 were obtained thanks to the `infnorm` (infinity norm) command: it computes an upper bound on the approximation error $\delta_{\text{approx}}(X) = |f(X) - p(X)|$ over the interval I or in mathematical terms

$$\bar{\delta}_{\text{approx}} = \max_{x \in I} |f(x) - p(x)|$$

The corresponding Sollya command is

```
sup(infnorm(f-p, [0;1]));
```

As computation of the infinity norm is nontrivial [Che+11], `infnorm` returns a confidence interval, which is why the `sup` is required. There is also another Sollya function for this purpose, `supnorm`, which is more complex to use but more robust.

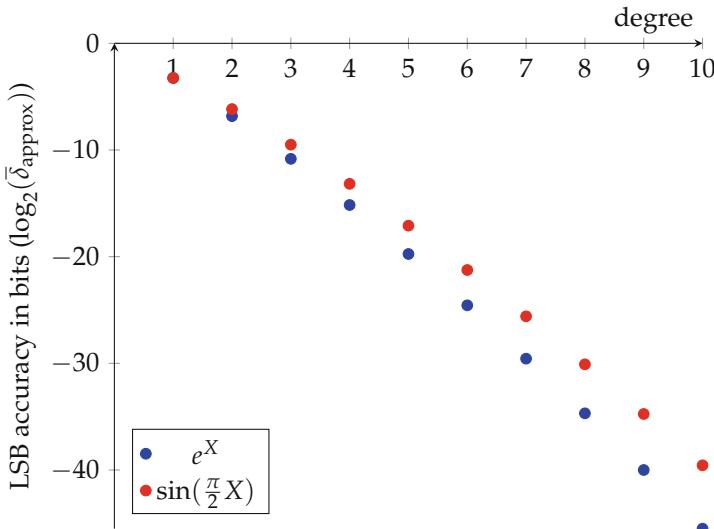


Fig. 18.4 Accuracy (in bits) of Remez approximations to two function on $[0, 1]$, with respect to the degree.

18.1.3 Relationship Between Degree and Accuracy

Figure 18.4 plots the accuracy of the Remez approximation with respect to the degree, for two functions. It shows that this accuracy depends on the function but is roughly linear in the degree. In short, for doubling the number of accurate bits, one should expect to double the degree.

Again, this is a rule of thumb which is very useful to reason about an architecture, but the actual value of the degree ensuring a given accuracy for a given function should be computed numerically.

18.1.4 Relationship Between Interval Size and Accuracy

There is one more degree of freedom that we have, and it is the interval itself. Figure 18.5 plots, for the two functions $f(X) = e^X$ and $f(X) = \sin(\frac{\pi}{2}X)$, the evolution of the LSB accuracy of $f(X)$ on two intervals of size 2^{-n} with respect to the integer n . We arbitrarily chose one interval near 0 and one interval near 1: the intervals are $[0, 2^{-n}]$ and $[1 - 2^{-n}, 1]$.

Here again, the rule of thumb is that this dependency is roughly linear. In other words, dividing the interval size by two improves the accuracy by a constant number of bits. Now, this constant number of bits (the slope of the various lines in Fig. 18.5) depends on the function but also on the location of the interval.

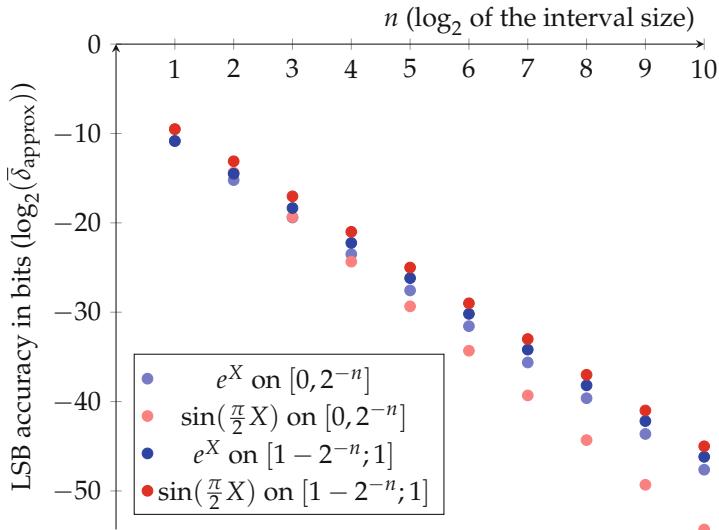


Fig. 18.5 Accuracy (in bits) of Remez degree-3 approximations to two function on intervals of size 2^{-n} , with respect to n .

How is this information relevant, since we chose to evaluate all our functions on either the interval $[0, 1]$ or the interval $[-1, 1]$? Again, it is a matter of a simple change of variable: the accuracy of $f(X)$ on $[0, 2^{-n}]$ is also the accuracy of $f(2^{-n}X)$ on $[0, 1]$. Similarly, the accuracy of $f(X)$ on $[1 - 2^{-n}; 1]$ is also the accuracy of $f(1 - 2^{-n} + 2^{-n}X)$ on $[0, 1]$.

This is what we are going to exploit in the sequel: Sect. 18.2 is dedicated to *range reduction* techniques which, as the name suggests, attempt to reduce the size of the interval in order to reduce the degree of the approximation, hence its cost.

Before this, we first address the important issue of *discretizing* the real coefficients of our polynomials.

18.1.5 Machine-Efficient Polynomial Approximation

The coefficients of a Remez approximation polynomial p_{Remez} are real numbers: they must be rounded to finite-precision machine numbers before we can use them in an architecture. Rounding the coefficients creates a new polynomial \tilde{p}_{Remez} .

A first option [Lee+05; Lee+09] is to take this into account in the error analysis as a new error term $\delta_{\text{round-coeff}}(x) = \tilde{p}_{\text{Remez}}(X) - p_{\text{Remez}}(X)$. A second, better option is to simply redefine the approximation error out of

\tilde{p}_{Remez} , i.e., $\delta_{\text{approx}}(X) = \tilde{p}_{\text{Remez}}(X) - f(X)$. This is simpler and provides a tighter error bound.

However, there is no mathematical reason why rounding the coefficients of the Remez polynomial should provide the best possible polynomial approximation among the set of polynomials whose coefficients are machine numbers. In other words, there may be better polynomials than \tilde{p}_{Remez} , i.e., polynomials with machine-representable coefficients that provide a better approximation (a smaller error).

Indeed, an algorithm [BC07] available as the `fpmiminimax` command of the Sollya tool [CJL10] only considers polynomials whose coefficients satisfy some user-specified constraints on the number formats of the coefficients. These constraints may be, for instance, that all coefficients should be single-precision floating-point numbers. In our case (we are interested in building fixed-point approximator hardware for an absolute error bound), the relevant constraints are to specify the LSB position of each coefficient. The tool allows to specify each of these LSB values separately. It then provides a polynomial that is generally extremely close to a minimax among these constrained polynomials.

To define these constraints, consider a theoretical architecture evaluating the developed form of a polynomial $C_0 + C_1 X + C_2 X^2 + \dots + C_d X^d$ on $I = [0, 1]$ or $I = [-1, 1]$. It would first compute each monomial $C_i X^i$ and then align them to compute the fixed-point sum. Such an alignment is depicted in Fig. 18.6 for a polynomial evaluating $f(X) = e^X$ on $[0, 1]$ to an accuracy of 10^{-6} (it is close to the Taylor series).

As $|X| \leq 1$, we also have $|X^i| \leq 1$, hence $|C_i X^i| \leq |C_i|$. Therefore, the MSB of each monomial term $C_i X^i$ is that of C_i . This is actually the main reason for standardizing our interface on the two intervals $[0, 1]$ and $[-1, 1]$.

$C_0 = 0.111111111111111100101\dots$	≈ 1
$C_1 = 1.00000000000000001011010111\dots$	≈ 1
$C_2 = 0.10000000000000001010011011\dots$	$\approx 1/2$
$C_3 = 0.0010101010011110000010000\dots$	$\approx 1/6$
$C_4 = 0.0000101010100111111011010\dots$	$\approx 1/24$
$C_5 = 0.0000001000110010100110001\dots$	$\approx 1/120$
$C_6 = 0.0000000001011110101111001\dots$	$\approx 1/720$
$C_0 \quad 0.111111111111111100101\dots$	
$+C_1 X \quad x.xxxxxxxxxxxxxxxxxxxxxxx\dots$	
$+C_2 X^2 \quad 0.xxxxxxxxxxxxxxxxxxxxxxx\dots$	
$+C_3 X^3 \quad 0.0xxxxxxxxxxxxxxxxxxxxxx\dots$	
$+C_4 X^4 \quad 0.000xxxxxxxxxxxxxxxxxxxxxx\dots$	
$+C_5 X^5 \quad 0.0000000xxxxxxxxxxxxxx\dots$	
$+C_6 X^6 \quad 0.000000000xxxxxxxxxxxxxx\dots$	

Fig. 18.6 Remez approximation (here to e^X) for $X \in [-1, 1]$: the alignment of the $C_i X^i$ follows that of the C_i .

Let us now discuss the LSBs to derive the constraints to input to `fpmiminimax`, assuming that we have an accuracy objective $\bar{\delta}_{\text{approx}}^{\text{target}}$. First, there is no point in evaluating one of the monomials much more accurately than the others, as the sum will be no more accurate than its least accurate term. Besides, for X close to 1, the monomial $C_i X^i$ cannot be more accurate than C_i itself. A near-optimal design decision is therefore to have each C_i accurate to the same LSB ℓ_p . The lower ℓ_p , the wider the numbers manipulated by the architecture; therefore, it is desirable to find the maximal value of ℓ_p that allows us to reach the accuracy objective $\bar{\delta}_{\text{approx}}^{\text{target}}$.

Intuitively, we need $2^{\ell_p} \leq \bar{\delta}_{\text{approx}}^{\text{target}}$ so the coefficients hold information that is at least as accurate as $\bar{\delta}_{\text{approx}}^{\text{target}}$. In practice, surprisingly, there usually exist polynomials satisfying $\bar{\delta}_{\text{approx}} \leq \bar{\delta}_{\text{approx}}^{\text{target}}$ even when 2^{ℓ_p} is slightly larger than $\bar{\delta}_{\text{approx}}^{\text{target}}$. This effect improves with the degree d ; therefore, a good heuristic is to start by calling `fpmiminimax` with $\ell_p = \lceil \log_2(\bar{\delta}_{\text{approx}}^{\text{target}}) \rceil + d$. If `fpmiminimax` fails to find a polynomial that satisfies $\bar{\delta}_{\text{approx}} < \bar{\delta}_{\text{approx}}^{\text{target}}$, then ℓ_p is decreased until it succeeds. This loop (formalized in Algorithm 18.1) rarely needs more than a few attempts, and its execution time remains well below one second. It is implemented in FloPoCo in the `BasicPolyApprox` class. The computation of $\bar{\delta}_{\text{approx}}$ in this loop is performed with Sollya `supnorm(p, f, I)` command [Che+11].

Of course, a condition for this loop to terminate is that d is such that the degree- d Remez approximation (which somehow corresponds to $\ell_p \rightarrow -\infty$) achieves an absolute error strictly smaller than $\bar{\delta}_{\text{approx}}^{\text{target}}$.

One final remark on this subject, Algorithm 18.1 is a heuristic. It is possible to relax the choice made of having the same LSB value for all the coefficients and save a bit here and there. In particular, allowing more LSBs for the constant coefficient C_0 will be almost for free in terms of evaluation, as C_0 only participates to one addition, but not to a product (see Figs. 18.19 and 18.20, for instance). If it is possible to trade more bits of C_0 for fewer bits on the other coefficients, it should be a win.

Such refinements should be directed by the economy of hardware that they entail: it is better to discuss them once the polynomial evaluation scheme has been chosen, which is the subject of Sect. 18.3.

18.1.6 Summing Up

To summarize, among the various approximation techniques that have been studied in the literature, two will be of practical relevance:

- Machine-efficient polynomial approximation can be provided by Sollya's `fpmiminimax`.

Algorithm 18.1: A heuristic for fixed-point polynomial approximation on $I = [0, 1]$ or $I = [-1, 1]$ minimizing coefficient sizes. It assumes that there exists a Remez polynomial of degree d with error smaller than $\bar{\delta}_{\text{approx}}^{\text{target}}$

```

function FixedPointPolyApprox( $f, d, I, \bar{\delta}_{\text{approx}}^{\text{target}}$ )
     $\ell_p \leftarrow \lceil \log_2(\bar{\delta}_{\text{approx}}^{\text{target}}) \rceil + d + 1$ 
    repeat
         $\ell_p \leftarrow \ell_p - 1$ 
         $p \leftarrow \text{fpminimax}(f, d, I, \{\ell_p\})$ 
         $\bar{\delta}_{\text{approx}} \leftarrow \text{supnorm}(p, f, I)$ 
    until  $\bar{\delta}_{\text{approx}} \leq \bar{\delta}_{\text{approx}}^{\text{target}}$ 
    return  $(p, \bar{\delta}_{\text{approx}}, \ell_p)$ 

```

$C_0 = 1.101001011101111000$	$C_0 = 1.1010011000010010011$
$C_1 = 0.110100101111011000$	$C_1 = 0.0011010011000010010$
$C_2 = 0.0011010111011101111$	$C_2 = 0.0000001101001101001$
$C_3 = 0.0000100011110101100$	$C_3 = 0.00000000000100011001$
$f\left(\frac{X}{2} + 0.5\right)$ on $[-1, 1]$	$f\left(\frac{X}{16} + 0.5\right)$ on $[-1, 1]$

Fig. 18.7 Two degree-3 fpminimax approximations to $f(X) = e^X$.

- Taylor approximation may in some cases provide simpler polynomials for the same degree (thanks to simple coefficients such as zero or one), which can in such cases compensate its larger worst-case approximation error.

The rules of thumb regarding accuracy versus degree and/or interval size that were observed on Remez polynomials in Sects. 18.1.3 and 18.1.4 also apply to fpminimax polynomials. In particular, a change of variable corresponding to a reduction of the interval size ($X \mapsto X/k$), for the same degree, also has the effect of reducing each coefficient of the corresponding approximation polynomial. This is illustrated by Fig. 18.7. This effect is important to understand because, in the sequel, the datapath will be dimensioned to match the sizes of the coefficients.

18.1.7 Polynomials Are a Special Case of Rational Fractions

A rational fraction is defined as the quotient of two polynomials:

$$R(x) = \frac{P_0 + P_1x + P_2x^2 + \cdots + P_nx^n}{1 + Q_1x + Q_2x^2 + \cdots + Q_mx^m} \quad \text{with } P_i, Q_i \in \mathbb{R} . \quad (18.2)$$

Polynomials are the special case when $m = 0$.

Most of the techniques presented above for polynomial approximation can be extended to the rational case. For instance, the Remez implementation in most mathematical package can determine a rational approximation [Mul16]. However, the present book mostly ignores rational approximations for the following reason:

For functions without singularities, a rule of thumb [BM04] seems to be that a rational approximation of degree n on the numerator and m on the denominator provides roughly the same accuracy as a polynomial approximation of degree $n + m$. Since the former requires the same number of multiplications as the latter, plus an expensive division, it is not an attractive option, unless the division is fused with the multiplications as in the E-method that will be presented in Sect. 19.4.

Some useful functions have singularities on their domain of interest: tangent goes to infinity, square root has an infinite derivative in 0, etc. For such functions, a rational approximation looks attractive since it has the potential to capture the singularities (e.g., as zeroes of the denominator). However, in the experience of the authors, it is often a false good idea. In some cases, the singularity can be avoided by some range reduction that will turn out cheaper to implement than division (this is the case of the square root, see Sect. 19.1). In other cases, the division is unavoidable; therefore, an architecture will involve a divider, but then it is better to explicit it in the construction of the algorithm. For instance, $\tan(x)$ is $\sin(x)$ divided by $\cos(x)$, and it will be productive to share a range reduction between the sine and the cosine of the same value, an opportunity that could be lost if blindly computing a rational approximation.

An argument in favor of rational approximation could be that the numerator and denominator can be evaluated in parallel, thus potentially allowing for a lower latency. The latency of division usually voids this argument. Besides, there are polynomial evaluation techniques that provide such parallelism without division and with more flexibility: see the Estrin schemes introduced below in Sect. 18.3.3.

All this discussion is more a summary of experience of the authors than definitive science. There probably exist functions for which the best implementation would be based on rational approximations. What is currently missing is a solid implementation that systematically compares state-of-the-art polynomial and rational approximations for a given context.

18.2 Generic Range Reduction Techniques

18.2.1 Range Reduction by Uniform Piecewise Splitting

The uniform segmentation scheme [Lee+05; DJP10] is similar to what was used in the bipartite method. The input interval is split into 2^α subintervals of identical size (Fig. 18.8). An approximation polynomial can then be used on each interval. We thus have 2^α polynomials, whose coefficients must be stored in a table. The index to this table is an integer $A \in \{0 \dots 2^\alpha - 1\}$, formed by extracting the α most significant bits of the input (Fig. 18.9). The bit vector Y is composed of the remaining $w_X - \alpha$ least significant bits. It represents the offset of X within an interval: it is input to a polynomial evaluator (Fig. 18.10) that also inputs the coefficients. Figures 18.19 and 18.20 are examples of polynomial evaluator architectures.

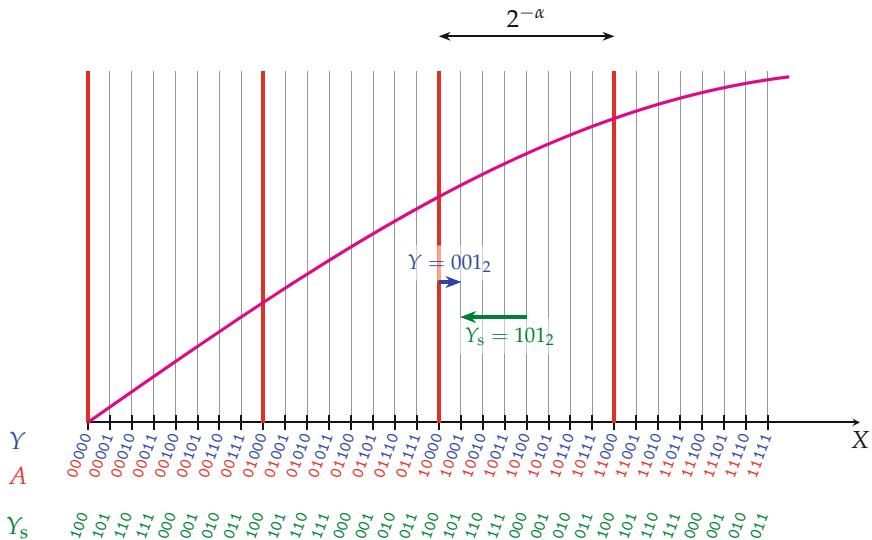


Fig. 18.8 Uniform segmentation of the input domain ($\alpha = 2$).



Fig. 18.9 Input word decomposition for uniform segmentation.

18.2.1.1 The Memory/Arithmetic Trade-Off

In this scheme, the parameter α controls a trade-off between memory and arithmetic resources: a larger α means a larger number of smaller intervals. Smaller intervals mean that a lower degree can be used to reach a given accuracy; hence, fewer multipliers will be needed. A lower degree also means fewer coefficients, hence less memory per polynomial. However, this effect is dwarfed by the fact that the number of polynomials to store is 2^α : the memory requirement still grows almost exponentially with α . In short, a larger α means more memory but fewer multipliers.

The user of a polynomial generator should be in control of this memory/arithmetic trade-off. For instance, a generator (Fig. 18.1) may input the degree d and determine the smallest possible α that ensures last-bit accuracy. This is the interface offered by FloPoCo. An alternative would be to input α and deduce the smallest possible d . One could also imagine a higher-level memory/arithmetic control knob.

Whatever the parameter to be determined, α or d , it can only take very few values. Therefore, an algorithm based on trial-and-error using increasing values will complete in a matter of seconds for all practical sizes.

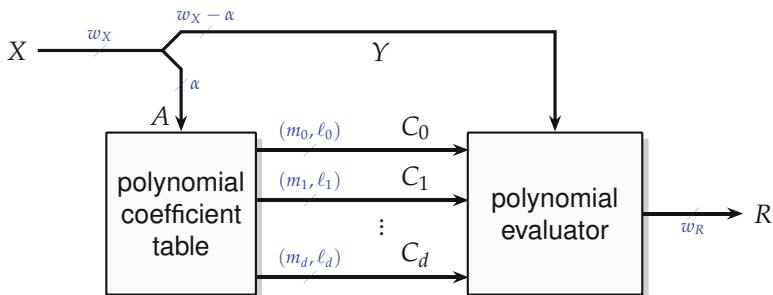


Fig. 18.10 Uniform segmentation architecture – see Figs. 18.19 and 18.20 for examples of polynomial evaluators.

18.2.1.2 Should the Reduced Argument Be Unsigned or Signed?

Remark that in this process, the signedness of X (whether it belongs to the interval $[0, 1]$ or to the interval $[-1, 1]$) is irrelevant. For simplicity, Fig. 18.8 and all the following address the case of unsigned X , but everything can be trivially adapted to signed X . How we interpret Y , however, happens to have an architectural impact, which is discussed now.

The simplest idea, used in most previous works [PBM01; Lee+05; OS05; Lee+09; DJP10; Che15; De +17], is to use the change of variable

$$X = 2^{-\alpha}(A + Y) \quad \text{for } Y \in [0, 1] \quad \text{and} \quad A \in \{0, 1, \dots, 2^\alpha - 1\} \quad . \quad (18.3)$$

See Fig. 18.8 for an example – note that we interpret the bits of A as an integer, since it will be used as an address/index. The A -th function, normalized on $[0, 1]$, is

$$f_{u,A}(Y) = f(2^{-\alpha}(A + Y)) \quad . \quad (18.4)$$

The reduced argument Y is obtained as the LSBs of X as illustrated in Fig. 18.8.

An alternative is to center the reduced argument on its subinterval: the change of variable is then

$$X = 2^{-\alpha}\left(A + \frac{1}{2} + \frac{Y_s}{2}\right) \quad \text{for } Y_s \in [-1, 1] \quad \text{and} \quad A \in \{0, 1, \dots, 2^\alpha - 1\} \quad . \quad (18.5)$$

The A -th function, normalized on $[-1, 1]$, becomes

$$f_{s,A}(Y_s) = f\left(2^{-\alpha}\left(A + \frac{1}{2} + \frac{Y_s}{2}\right)\right) \quad . \quad (18.6)$$

In general, this leads to smaller coefficients for the approximation polynomials, as illustrated by Fig. 18.11. The reason for this is essentially that the argument is more reduced in (18.6) than in (18.4).

The architectural overhead to obtain Y_s out of Y is limited to one inverter to complement the MSB of Y , as Fig. 18.8 illustrates. To the author’s understanding, the constrained polynomials of [SDP11] are essentially another (more contrived) way to express the same thing.

18.2.1.3 Coefficient Sizing

For each A , i.e., for each subinterval, Algorithm 18.1 can be used to determine an approximation polynomial with its coefficients $C_i(A)$, each with its fixed-point format $(m_i(A), \ell_i(A))$. To build the architecture of Fig. 18.10, the generator then needs to dimension the wire that can hold all these coefficients, i.e., determine the fixed-point formats (m_i, ℓ_i) visible in Fig. 18.10. This is a simple pass computing the worst-case format for each coefficient, i.e.,

$$\forall i \quad m_i = \max_{\forall A} m_i(A) \quad (18.7)$$

$$\ell_p = \min_{\forall A} \ell_i(A) \quad (18.8)$$

(in Algorithm 18.1, all the ℓ_i are chosen equal and called ℓ_p). For most of the polynomials, the common ℓ_p is lower than the $\ell_p(A)$ that had been deter-

Range reduction to unsigned:	$C_0 = 1.110111001000101011$
$f_{u,10}(Y) = \exp\left(\frac{10}{16} + 2^{-4}Y\right)$	$C_1 = 0.000111011100100010$
with $Y \in [0, 1]$	$C_2 = 0.0000000001110111011$
	$C_3 = 0.000000000000000100111$
Range reduction to signed:	$C_0 = 1.1110110101110100000$
$f_{s,10}(Y_s) = \exp\left(\frac{10.5}{16} + 2^{-5}Y_s\right)$	$C_1 = 0.000011101101011101$
with $Y_s \in [-1, 1]$	$C_2 = 0.00000000000111101101$
	$C_3 = 0.000000000000000101$

Fig. 18.11 Range reduction with signed reduced arguments entails smaller polynomial coefficients. Here, $f_{u,10}$ and $f_{s,10}$ cover the same subinterval of $\exp(x)$, and both polynomials are accurate to 10^{-6} .

mined by Algorithm 18.1. It is therefore possible to invoke fpminimax one more time with this extended precision. This will improve the average error of the operator at no additional cost. Another option is to have the loop over A inside the main loop of Algorithm 18.1, so that all the polynomials share the same value ℓ_p .

18.2.1.4 Coefficient Compression Opportunities

Figures 18.12, 18.13, and 18.14 show plots of the value of each coefficient as a function of the interval index for two different functions. An important observation is that for these two regular functions, the coefficient plots remain very regular. Indeed, they would be perfectly smooth if the polynomials were Remez or Chebyshev ones. The artifacts due to the machine-efficient approximation become only observable on C_3 . More generally, they affect mostly the highest degree coefficient, as Fig. 18.14 illustrates.

This smoothness has an important consequence: The table compression used by Hsiao [Hsi+15] for the table holding C_0 in multipartite tables (see Sect. 17.2.8) can be used here also, and not only on C_0 . Each table of coefficient can be compressed without error at the cost of one addition. As was the case for multipartite table, this addition will add only a handful of bits to some bit heap, and its delay will be completely hidden most of the times, all the more as most coefficients are used late in polynomial evaluation hardware – see Sect. 18.3.

These plots may also suggest that a function-specific range reduction should be investigated. This potential is well illustrated by the sine function: Fig. 18.12 illustrates what its coefficients will look like. In this case, it seems obvious that all the coefficients could be computed out of the table of C_0 . This can be mathematically explained by the well-known identity $\sin(a + b) = \sin(a)\cos(b) + \cos(a)\sin(b)$. Applying it to (18.3) for the function $\sin(\frac{\pi}{2}X)$, we obtain

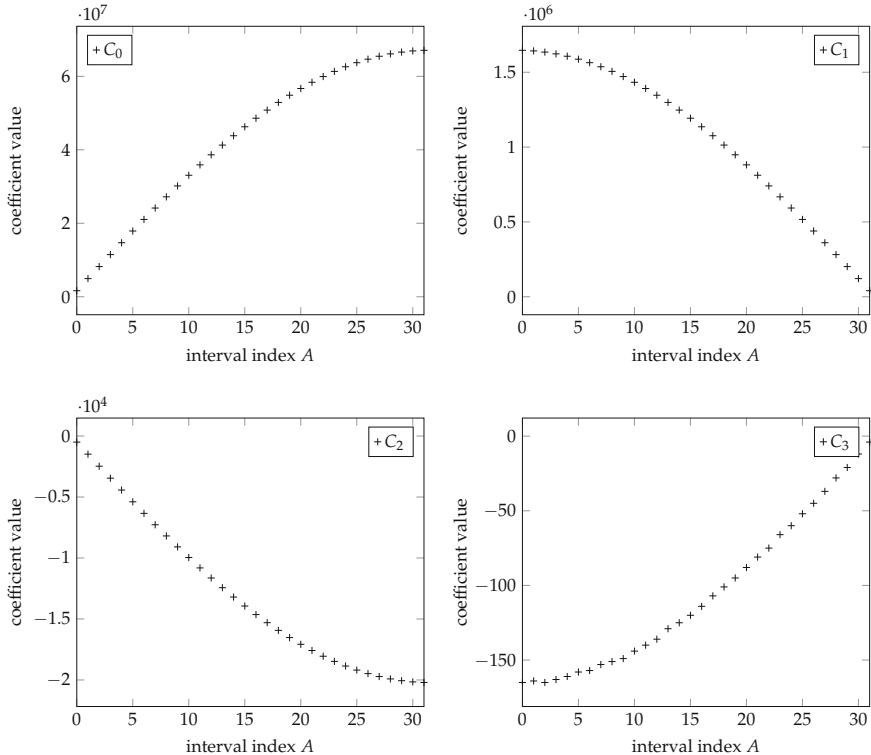


Fig. 18.12 Evolution of the coefficients over the subintervals for a 24-bit, degree-3 approximation to $\sin(\frac{\pi}{2}x)$.

$$\begin{aligned}
 \sin\left(\frac{\pi}{2}X\right) &= \sin\left(\frac{\pi}{2}2^{-\alpha}(A+Y)\right) \\
 &= \sin(a)\cos(b) + \cos(a)\sin(b) \\
 \text{with } a &= \frac{\pi}{2}2^{-\alpha}A \quad \text{and} \quad b = \frac{\pi}{2}2^{-\alpha}Y
 \end{aligned} \tag{18.9}$$

Now, $a \in [0, 1]$, while $b \in [0, 2^{-\alpha}]$: b is small; therefore, we may use a degree-3 Taylor approximation for its sine and cosine:

$$\sin(b) \approx b - \frac{b^3}{6} \tag{18.10}$$

and

$$\cos(b) \approx 1 - \frac{b^2}{2}. \tag{18.11}$$

Merging (18.10) and (18.11) into (18.9), we obtain a polynomial in Y whose coefficients are

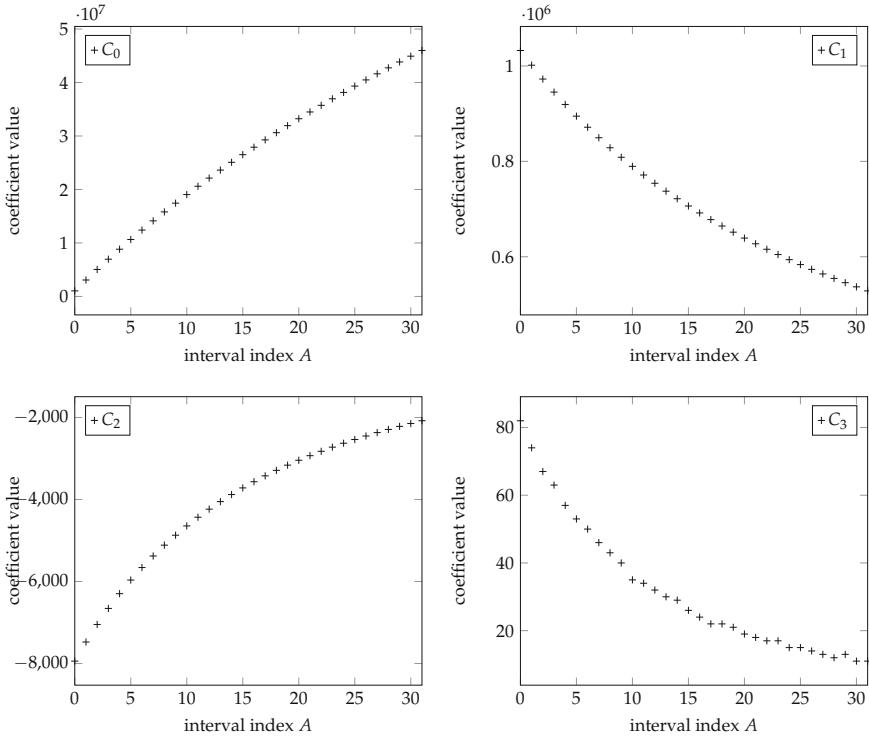


Fig. 18.13 Evolution of the coefficients over the subintervals for a 24-bit, degree-3 approximation to $\log(1 + x)$.

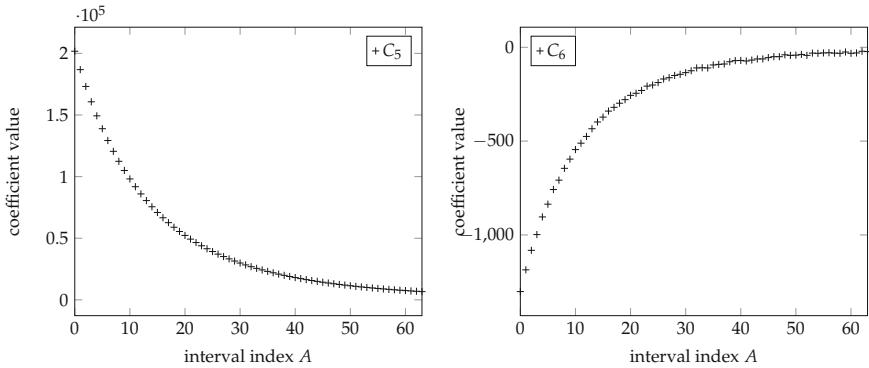


Fig. 18.14 Evolution of the coefficients over the subintervals for a 52-bit, degree-6 approximation to $\log(1 + x)$.

$$C_0(A) = \sin\left(\frac{\pi}{2}2^{-\alpha}A\right) \quad (18.12)$$

$$C_1(A) = \frac{\pi}{2}2^{-\alpha}\cos\left(\frac{\pi}{2}2^{-\alpha}A\right) \quad (18.13)$$

$$C_2(A) = -\frac{\pi^2}{2^2}\frac{2^{-2\alpha}}{2}C_0(A) \quad (18.14)$$

$$C_2(A) = \frac{\pi^3}{2^3}\frac{2^{-3\alpha}}{6}C_1(A) \quad (18.15)$$

Finally, we may reduce C_1 to C_0 using another trigonometric identity: $\cos(x) = \sin(\frac{\pi}{2} - x)$.

Chapter 20 will construct an architecture that specifically exploits these function-specific features. It actually tabulates both sine and cosine but on $[0, \pi/4]$: these two tables together correspond to a table of C_0 of Fig. 18.12.

18.2.1.5 Coefficients with Constant Signs

There is a side effect of the previous observation: it is typical (but not automatic due to numerical artifacts) that all the coefficients of a given degree have the same sign. In this case, the sign bit need not be stored, saving one bit per coefficient. In practice, the implementation does not need to perform any complex mathematical analysis, but should simply check if all the coefficients of a given degree have the same sign.

Another point of view is that some analytical properties of the function (monotonicity, convexity, and so on) are defined by the signs of the successive derivatives. Good approximation polynomials typically inherit these properties, which are then reflected in the signs of the corresponding coefficients.

18.2.2 Interpolation

Interpolation [Lee+08] is based on the tabulation of function values. A polynomial is used to evaluate the function between the tabulated values, but contrary to the previous methods, the polynomial coefficients are computed out of neighboring tabulated values.

For a hardware-efficient interpolation, the tabulated values must be regularly spaced. Then the architecture can be compared to a uniform segmentation architecture where only the degree-0 coefficients would be tabulated: the advantage would be to save the storage of the higher degree coefficients.

However, there are several issues with this technique:

- The computation of the coefficients requires additional hardware, which adds to the delay and area.

- This hardware scales poorly with the degree. Besides, higher degrees require the tabulation of more function values, so the cost of the table increases with the degree, along with the arithmetic cost. In contrast, in a uniform segmentation architecture, the cost of the polynomial coefficient table decreases with the degree while the arithmetic cost increases.
- For the interpolation polynomial to be accurate enough, the hardware computing the coefficients needs to be even more accurate.
- Interpolation is poorly suited to nonuniform range reductions.

These issues have been studied quantitatively [Lee+08], and the conclusion is essentially that tabulating polynomials (as studied so far in this chapter) is more efficient than interpolation. More precisely, according to [Lee+08], interpolation really offers some benefit in terms of area for degree-1 (linear) piecewise approximation with uniform segmentation. But then it has also been shown that the multipartite schemes are even more efficient in this case [DT05].

This being said, there certainly exist specific situations where interpolation makes sense, and it is a technique one should be aware of.

18.2.3 Logarithmic Piecewise Splitting

For some functions, uniform segmentation will lead to an unbalanced architecture. For instance, in Fig. 18.15, the variations of the derivative on the left of the interval entail that a higher degree is needed for a subinterval there, compared to a subinterval of the same size on the right of the figure. It is possible to use uniform segmentation, but with the same degree, the polynomials on the right will be vastly more accurate than needed.

In a logarithmic piecewise splitting, the index to the coefficient table is computed as the leading zero count on the input (if more accuracy is required on the left of the interval, as in Fig. 18.15) or as a leading one count (if more accuracy is required on the right). The reduced argument is composed of the remaining bits, which must be shifted into leading position. This is a case for a combined leading zero count and shifter (introduced in Sect. 10.3).

In the case depicted in Fig. 18.15, the change of variable defining the function to evaluate is $X = 2^{A-1}(1 + Y)$ with $Y \in [0, 1]$ the shifted value without the leading bit which is by definition a 1. For instance, for $X = .00101$, we have $A = 2$; X is therefore shifted left by 2, giving 10100; and dropping the leading one gives $Y = .01000$. The A -th function to approximate is

$$f_{u,A}(Y) = f\left(2^{-A-1}(1 + Y)\right). \quad (18.16)$$

Here again, it is slightly better to use a signed value for the reduced argument. The change of variable is in this case $X = 2^{-A-2}(3 + Y_s)$, and the A -th

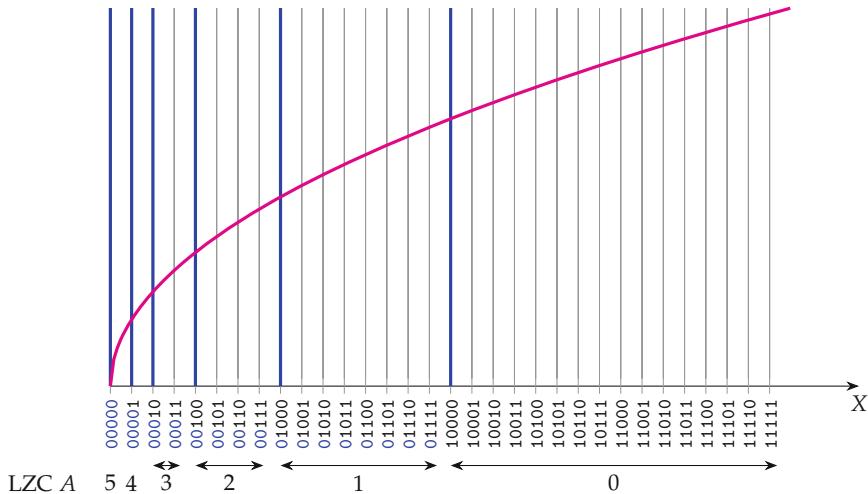


Fig. 18.15 Logarithmic segmentation, counting leading zeroes.

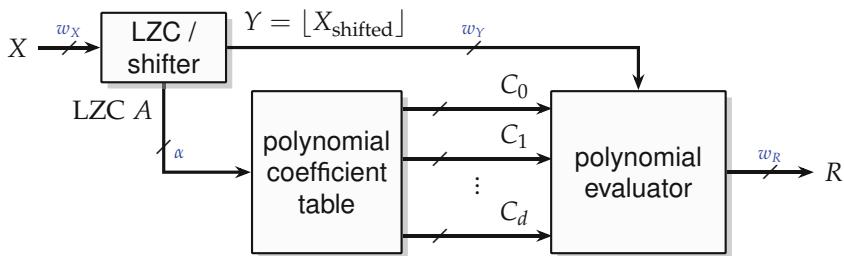


Fig. 18.16 Example of logarithmic segmentation architecture.

function becomes

$$f_{s,A}(Y_s) = f\left(2^{-A-2}(3 + Y_s)\right). \quad (18.17)$$

18.2.3.1 Size of the Reduced Argument

A drawback of this approach is that w_Y does not seem reduced: in the worst case (e.g., $A = 0$; see Fig. 18.15), there is no shift, so we keep all the bits of X – except the leading bit which is constant on the interval and actually removed by (18.17). Therefore, the multipliers in the polynomial evaluator will be larger than what they can be in a uniform segmentation, where Y is always of size $w_Y = w_X - \alpha$.

In practice, however, it is often possible to have $w_Y < w_X - 1$ by shaving a few of the lower significant bits from the shifted X . This is what is

expressed by $Y = \lfloor X_{\text{shifted}} \rfloor$ in Fig. 18.16. This can be intuitively understood by looking more closely at Fig. 18.15. In this example, in all the large interval corresponding to $A = 0$, the function has a small slope, which means that several consecutive input values will round to the same output value. These several input values differ only by their least significant bits: if we remove these bits, it may still be possible to achieve last-bit accuracy. Of course, if we remove k LSB bits, the impact is limited to the k rightmost intervals: on the other intervals, we are removing zero bits introduced to the right by the left shifter.

This effect is very function-dependent and also depends on the respective input and output resolutions (the above handwaving explanation assumed they were identical). In practice, it is best managed by considering w_Y as a parameter. The generator algorithm will initialize it at $w_Y = w_X - 1$ and then decrement it until the target accuracy can no longer be achieved.

Note that we have two intervals, at the left of Fig. 18.15, which are composed of only one point. There, the polynomial can reduce to a_0 with all other coefficients set to 0, which allows for an easy management of possible discontinuities of the function (the infinite derivative in the example of Fig. 18.15). Logarithmic segmentation is indeed relevant for functions which have discontinuities.

We leave as an exercise to the reader to adapt all the previous to two other main cases:

- when A is the leading one count, which should be attempted with functions with a discontinuity on the right of the interval,
- when A is the leading sign bit count, which should be attempted with functions with discontinuities on both sides of the interval.

18.2.3.2 Saturating the Leading Bit Count

So far, logarithmic segmentation entails no trade-off, and there is no parameter to control the size of the table or the number of multipliers. Besides, if the input size w_X is not a power of two (e.g., $w_X = 5$ in Fig. 18.15), we have a number of polynomials that is not a power of two, either. This is not very architecture-friendly. A solution to these two issues is to cap the value of the leading bit count to some power of two. In Fig. 18.16, we have one parameter, α , that defines the address size. By default, $\alpha = \lceil \log 2w_X \rceil$, but it is also possible to have a smaller value. For the example of Fig. 18.15, we could, for instance, use $\alpha = 2$. The maximum value of A would then be $A = 3$: what happens in this case is that the three leftmost intervals of Fig. 18.15 are merged into one, with one single approximation polynomial. Whether this works well or not again depends on the function: on the one side, we are close to the discontinuity, so polynomial approximation is expected to work poorly. On the other side, we use polynomial evaluator hardware that

is over-dimensioned on the left of the interval, since the reduced argument has only a few significant digits there, the rest being LSB zeroes introduced by the shift.

18.2.4 Hierarchical Segmentation

The next natural idea is to stack several levels of segmentation. This idea was introduced in a very generic way with an arbitrary number of stacked decomposition [Lee+03], but experience then showed [Lee+09] that only two situations are useful in practice:

UU: Two levels of uniform segmentation. Each subinterval of a uniform decomposition of parameter α_1 is itself uniformly decomposed in subintervals. The number of second-level intervals is a power of two that depends on the first-level interval. See Fig. 18.17 for an illustration. A smaller table indexed by A_1 holds for each first-level interval the parameters of the second-level decomposition: first, the parameter $\alpha_2(A_1)$ of the second level and, second, an offset $\Delta(A_1)$ which points to the address of the first polynomial of this first-level interval. Since $\alpha_2(A_1)$ is not constant, a shifter is needed to decompose Y_1 into an index A_2 and the final reduced argument Y_2 input to the polynomial evaluator.

LU: A first level of logarithmic segmentation captures the discontinuities of the function, followed by a level of uniform segmentation on each subinterval. Again, the number of second-level intervals depends on the first-level interval. The architecture is essentially similar to that of Fig. 18.17, except that the first-level decomposition into (A_1, Y_1) is performed thanks to a combined leading bit counter and shifter as per Fig. 18.16.

The LU hierarchical decomposition can also be viewed as a generic way to introduce parameters – the values of $\alpha_2(a_1)$ – controlling the memory/arithmetical trade-off in a logarithmic segmentation.

In the experiments reported [Lee+09], some functions perform better with LU; some perform better with UU. It should be noted that these experiments were performed with a polynomial evaluator where all the multipliers have the same size, which is inferior to the evaluator computing just right that we describe in Sect. 18.3.1. A careful reimplementation remains to be done.

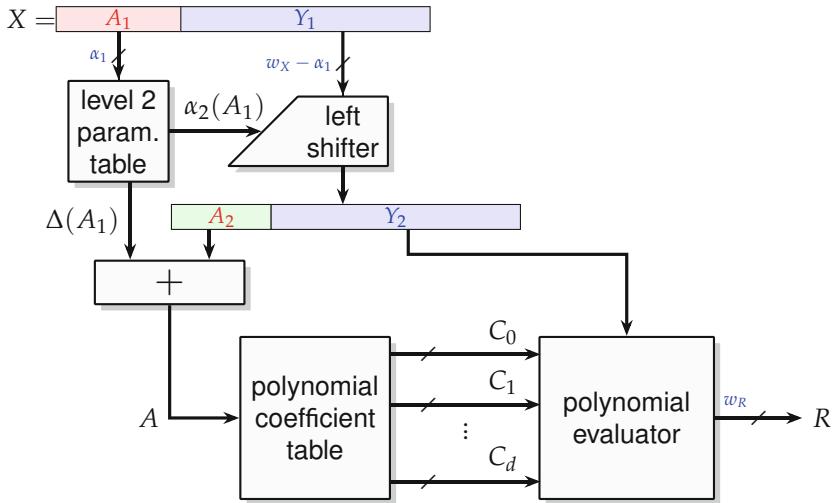


Fig. 18.17 Two-level uniform segmentation (UU) architecture.

18.2.5 Arbitrary Dichotomy-Based Segmentation

A last segmentation technique [SNB05; CK12; Tho15; KL14] is simply to decompose the input interval into a partition of subintervals that balance the approximation error for a given degree.

This is better from an approximation point of view, but the architecture (see Fig. 18.18) is more complex. Determining the index A to the polynomial table now requires a dichotomic search that uses either a tree of comparators or a sequence of table look-ups [SM04; SNB05] or another polynomial function that passes through the splitting points [KL14].

To reduce the cost of this dichotomic search, it is possible to base it only on the α most significant bits of the input. This entails that the approximation error is not strictly balanced between subintervals, but it has no impact in practice as long as all the approximation errors are comparable, e.g., within a factor 2 of each other.

The dichotomy process also outputs the middle M_A of each subinterval, and the reduced argument is computed as $Y = X - M_A$. The number of bits w_Y that is eventually input to the polynomial evaluator is also usually smaller than w_X but larger than $w_X - \alpha$. This is very function- and degree-dependent.

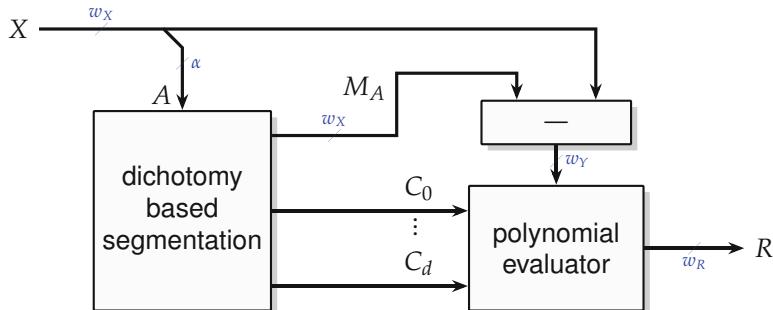


Fig. 18.18 Arbitrary decomposition architecture.

18.2.6 Discussion

Here, one really has to distinguish between two use cases for function approximators.

If the objective is to build a universal polynomial approximator, where the function itself is an input that is unknown when writing the function generator code, then each of the previous methods (uniform segmentation, logarithmic segmentation, hierarchical segmentation, arbitrary segmentation) is relevant. Indeed, for each of these methods, there exists a function for which this method gives the best results. Besides, the various methods may exhibit different trade-offs between latency and area, for instance, or between memory resources and logic resources. The universal, optimal function approximator remains to be written, and it will probably involve a lot of trial and error.

If the objective is to implement a given function that is known when writing the architecture generator code (e.g., sine/cosine in Chap. 20 or exponential in Chap. 22), then it is better to first consider the known properties of the function. This will usually help select the best range reductions and even adapt it to the function at hand.

For instance, Fig. 18.15 actually plots the square root function. This function obeys the identity

$$\sqrt{2^{2p}x} = 2^p\sqrt{x}. \quad (18.18)$$

From this identity, it seems natural to convert a fixed-point input X to the quasi floating-point representation $X = 2^{2p}Y$ using a leading zero counter and shifter similar to that shown in Fig. 18.16. Then the reduced argument Y begins either with 10 or with 01 and therefore belongs to the two rightmost segments of Fig. 18.15. We can have two polynomials there to evaluate it and finally a last shifter by p positions to bring it to place. We almost get the architecture of Fig. 18.16, with polynomial coefficient memory traded for a shifter. In [Din+10] (which addresses square root in floating point, not fixed point, which removes the need for the shifters), a uniform segmentation is

used on each of the two segments: this is effectively a hierarchical segmentation architecture but, again, very specialized to the square root.

Two other examples will be seen in Chaps. 20 and 22 for sine/cosine and exponential, respectively. In both cases, there is an ad hoc table-based range reduction which can be seen as a variation on the generic ones given in this chapter but adapted to the specifics of the function.

Still, in some cases, although the function is mathematically well defined, no useful range reduction can be found. In such situations, the generic techniques are relevant. This is, for instance, the case for most inverse cumulative distribution functions² [Che+07].

18.3 Hardware Polynomial Evaluation

This section addresses the construction of the box labeled “polynomial evaluation” in the Figs. 18.10, 18.16, 18.17, and 18.18.

Given a polynomial $p(Y) = \sum_{i=0}^d C_i Y^i$, there are many possible ways to evaluate it, with various cost-delay trade-offs. Section 18.3.1 will review an architecture based on the Horner evaluation scheme. It scales very well but has a long latency. Section 18.3.2 will review more parallel variants.

Each architecture also entails some *evaluation error* which adds to the approximation error studied so far. The evaluation error is the result of all the roundings that happen during the computation. The purpose of this section is also to show how to compute this error. Remember, however, that the coefficients C_i of the polynomials are already rounded and this is accounted for in the approximation error.

18.3.1 Horner Evaluation

The classic Horner evaluation scheme minimizes the number of operations at the expense of latency. It uses the following expression for $p(Y)$:

$$p(Y) = C_0 + Y \times (C_1 + Y \times (C_2 + \dots + Y \times C_d) \dots) \quad . \quad (18.19)$$

The error analysis in this case is based on expressing the Horner scheme as a recurrence:

² For the specific case of the Gaussian inverse cumulative distribution function or probit function, the asymptotic behavior near 0 suggests a logarithmic segmentation [JP20].

$$\begin{cases} S_i = \begin{cases} C_i + Y \times S_{i+1} & \forall i \in \{0 \dots d-1\} \\ C_d & \text{for } i = d \end{cases} \\ p(Y) = S_0 \end{cases} \quad (18.20)$$

As written here, (18.20) computes the exact value $p(Y)$ of the polynomial. This exact value would require a very accurate format and therefore be expensive to compute. In an architecture computing just right, we want to keep the intermediate formats, i.e., the format of the S_i , to the minimum. For this, we need to round the computations (essentially the products), so the values computed by the architecture will be some $\tilde{S}_i \approx S_i$. To formalize this, let us first introduce a new architectural parameter: let ℓ_i^S be the LSB position of the format of \tilde{S}_i .

Less intuitively, it is also beneficial to truncate or round the input Y at each step i to some \tilde{Y}_i . This will avoid inputting bits to multipliers which mostly influence product bits that will be rounded away (see Chap. 8 for a generic discussion of products computing just right). The corresponding architectural parameter is ℓ_i^S , the LSB position of the format of \tilde{S}_i .

The actual recurrence used is therefore the following:

$$\tilde{S}_i = \begin{cases} [C_i + \tilde{Y}_i \times \tilde{S}_{i+1}]_{\ell_i^S} & \forall i \in \{0 \dots d-1\} \\ C_d & \text{for } i = d \end{cases} \quad (18.21)$$

where the notation $[X]_\ell$ denotes some quantization of the value X to a format with LSB ℓ . The actual quantization used, e.g., rounding or truncation, is an architectural choice that will be discussed in the following. Figure 18.19 shows the corresponding architecture with these notations.

The cumulated evaluation error at step i is defined as

$$\begin{aligned} \delta_{\text{eval},i}(Y) &= \tilde{S}_i - S_i \\ &= [C_i + \tilde{Y}_i \times \tilde{S}_{i+1}]_{\ell_i^S} - (C_i + Y \times S_{i+1}). \end{aligned} \quad (18.22)$$

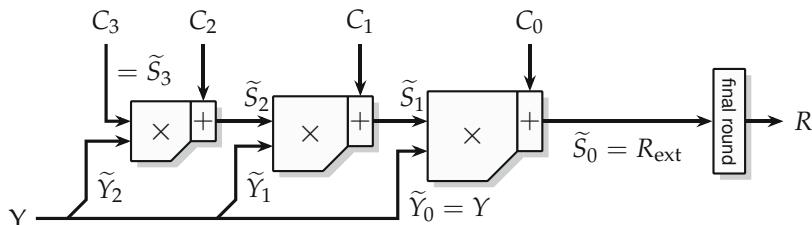


Fig. 18.19 A degree-3 Horner polynomial evaluator to be used with one of the previous range reduction schemes.

It can be computed following the methodology introduced in Chap. 3, which consists in rewriting (18.22) to isolate the various sources of errors by introducing the corresponding intermediate computation steps:

$$\begin{aligned}\delta_{\text{eval},i}(Y) &= \left[C_i + \tilde{Y}_i \times \tilde{S}_{i+1} \right]_{\ell_i^S} - (C_i + \tilde{Y}_i \times \tilde{S}_{i+1}) \\ &\quad + (C_i + \tilde{Y}_i \times \tilde{S}_{i+1}) - (C_i + Y \times \tilde{S}_{i+1}) \\ &\quad + (C_i + Y \times \tilde{S}_{i+1}) - (C_i + Y \times S_{i+1})\end{aligned}\quad (18.23)$$

which simplifies into

$$\begin{aligned}\delta_{\text{eval},i}(Y) &= \left[C_i + \tilde{Y}_i \times \tilde{S}_{i+1} \right]_{\ell_i^S} - (C_i + \tilde{Y}_i \times \tilde{S}_{i+1}) \\ &\quad + (\tilde{Y}_i - Y) \times \tilde{S}_{i+1} \\ &\quad + Y \times \delta_{\text{eval},i+1}.\end{aligned}\quad (18.24)$$

In (18.24), the first line corresponds to the arithmetic rounding of the operation itself. Let us define

$$\delta_{\text{multadd},i} = \left[C_i + \tilde{Y}_i \times \tilde{S}_{i+1} \right]_{\ell_i^S} - (C_i + \tilde{Y}_i \times \tilde{S}_{i+1}). \quad (18.25)$$

Now, C_i is already quantized to some format of LSB ℓ_i , defined, for instance, by (18.8). In the following, we may assume that the format of the result \tilde{S}_i is wider than the format of C_i , in particular $\ell_i \geq \ell_i^S$ (otherwise, it would mean that some bits of C_i are not really useful to the computation and the architecture could save their storage). For this reason, the addition is exact, and the arithmetic error essentially comes from the multiplication:

$$\delta_{\text{multadd},i} = \left[\tilde{Y}_i \times \tilde{S}_{i+1} \right]_{\ell_i^S} - \tilde{Y}_i \times \tilde{S}_{i+1}. \quad (18.26)$$

It can be bounded by $\bar{\delta}_{\text{multadd},i} = 2^{\ell_i^S}$ or $\delta_{\text{multadd},i} = 2^{\ell_i^S-1}$, depending on the multiplication technique used (truncation or rounding).

The second line of (18.24) corresponds to the truncation of Y . Let us define

$$\delta_{Y,i} = \tilde{Y}_i - Y. \quad (18.27)$$

Note that $\delta_{Y,i}$ may be null if there is no truncation at this step, i.e., if all the bits of Y are used. This should typically be the case for the last Horner step, where $\tilde{Y}_0 = Y$; otherwise, there is something wrong as the Horner evaluator inputs bits that are never used.

In the second line of (18.24), $\delta_{Y,i}$ is amplified by the term \tilde{S}_{i+1} itself. We will have to compute a bound \bar{S}_i on \tilde{S}_i to bound this error term. This bound will depend on the coefficients. In particular, in an architecture computing

a piecewise approximation, the evaluation error bound will depend on the subinterval.

The third line of (18.24) captures the propagation of the rounding error through the recurrence. It is also amplified by the value of Y , but this is easy to manage since Y is bounded in absolute value by 1. Thus, by triangle inequality on (18.24), we obtain the following bound:

$$\bar{\delta}_{\text{eval},i} = \bar{\delta}_{\text{multadd},i} + \bar{S}_{i+1}\bar{\delta}_{Y,i} + \bar{\delta}_{\text{eval},i+1} . \quad (18.28)$$

Since $\tilde{S}_d = C_d = S_d$, we have $\bar{\delta}_{\text{eval},d} = 0$; therefore, by recurrence,

$$\bar{\delta}_{\text{eval}} = \bar{\delta}_{\text{eval},0} = \sum_{i=0}^{d-1} (\bar{\delta}_{\text{multadd},i} + \bar{S}_{i+1}\bar{\delta}_{Y,i}) . \quad (18.29)$$

As usual, an architecture computing just right should balance the $2d$ error terms in (18.29). As already mentioned, the term $\bar{\delta}_{\text{multadd},i}$ will be either $2^{\ell_i^S}$ or $2^{\ell_i^S-1}$, depending on the quantization. We now assume that truncation is used for Y to get \tilde{Y}_i (rounding it to the nearest would cost one adder and save one bit in the multiplier input, thus saving hardware more or less equivalent to one adder). Therefore, we have $\bar{\delta}_{Y,i} = 2^{\ell_i^Y}$ if there was a truncation and $\bar{\delta}_{Y,i} = 0$ if there was no truncation. The architecture optimization algorithm therefore consists in evaluating (18.29) for formats of increasing sizes (decreasing ℓ_i^S) until $\bar{\delta}_{\text{eval}}$ is smaller than the target error budget. A good starting point for the ℓ_i^S is the LSB ℓ_i of the coefficients C_i , as determined during the approximation. In each step of the evaluation of (18.29), ℓ_i^Y is chosen such that $\bar{S}_{i+1}\bar{\delta}_{Y,i} \approx \bar{\delta}_{\text{multadd},i}$.

An example heuristic for this is presented in Algorithm 18.2, for the simpler case of a single polynomial. It is heuristic in the sense that all the Horner steps share the same LSB position: in many cases, a bit could be saved here and there by relaxing this constraint.

In the case of a piecewise approximation, Algorithm 18.2 is applied independently on each subinterval, computing the architectural parameters that suit this subinterval. The parameters for a polynomial that works on all the intervals are simply the worst-case ones (the max for the MSB positions, the min for the LSB positions). The reader interested in the details may find them in the FloPoCo source code.

Algorithm 18.2 inputs an error budget $\bar{\delta}_{\text{eval}}^{\text{target}}$. This value must be determined by a global error analysis combining approximation error and rounding error. This will be the object of Sect. 18.4 of this chapter.

Algorithm 18.2: An example heuristic for determining fixed-point formats for Horner evaluation computing just right

```

function FixedPointPolyEval (( $C_i$ ),  $\bar{\delta}_{eval}^{target}$ ,  $\ell_Y$ )
  for  $i = d$  down to 0 do
    | Compute  $\bar{S}_i$                                 // e.g., using Sollya interval arithmetic
  end for
   $\ell_S \leftarrow \ell_p$       // the LSB of the  $C_i$ , computed e.g., by Algorithm 18.1
  repeat
    |  $\bar{\delta}_{eval} \leftarrow 0$ 
    | for  $i = d - 1$  down to 0 do
      |   |  $\bar{\delta}_{multadd,i} = 2^{\ell_S^i}$  or  $2^{\ell_S^i - 1}$       // depending on multiplier used
      |   |  $\ell_{Y,i} \leftarrow \max(\ell_S - m_{\bar{S}_i}, \ell_Y)$ 
      |   |  $\bar{\delta}_{Y,i} = 2^{\ell_Y^i}$  if  $\ell_{Y,i} > \ell_Y$ , else 0
      |   |  $\bar{\delta}_{eval} = \bar{\delta}_{eval} + \bar{\delta}_{multadd,i} + \bar{S}_i \bar{\delta}_{Y,i}$ 
    | end for
    | if  $\bar{\delta}_{eval} < \bar{\delta}_{eval}^{target}$  then
    |   | done  $\leftarrow$  True
    | else
    |   |  $\ell_S \leftarrow \ell_S - 1$ 
    |   | done  $\leftarrow$  False
    | end if
  until done
  return ( $\ell_S, \{\ell_{Y,i}\}_{i \in \{0\dots d-1\}}$ )

```

18.3.2 Parallel Evaluation

Parallel evaluation uses the developed form of the polynomial:

$$p(Y) = C_0 + C_1 Y + C_2 Y^2 + \dots + C_d Y^d . \quad (18.30)$$

If computed in software, it requires $1 + 2 + \dots + d = d(d + 1)/2$ multiplications and d additions and is therefore more expensive than the Horner scheme which only requires d multiplications and d additions. However, it exposes more parallelism: Firstly, all the d power terms Y^i can be computed in $O(\log d)$ time by first computing the Y^{2^k} in sequence for $2^k \leq d$ and then multiplying them together, following to the binary decomposition of each i , to compute all the Y^i . Secondly, the sum of the $d + 1$ terms $C_i Y^i$ can be computed by an adder tree in $O(\log d)$ time as well. Thus, the whole polynomial, assuming enough multipliers are used, can be computed in $O(\log d)$ time, which is better than the $O(d)$ time of the Horner scheme.

However, these operation counts are not really relevant when generating hardware. What makes parallel evaluation interesting, especially in the context of a range reduction, is a combination of three observations:

- After a range reduction such as the uniform interval piecewise decomposition, the bit size of the coefficients C_i decreases with i (see, for instance, the coefficients in Fig. 18.7). Hence, the cost overhead of computing Y^i is offset by the fact that it is possible to use a truncated version \tilde{Y}_i of Y . Roughly speaking, an architecture computing just right will quantize $(\tilde{Y}_i)^i$, hence \tilde{Y}_i , to the same size as C_i . This observation is also true for some function-specific range reductions, such as those used for sine/cosine in Chap. 20 or the exponential in Chap. 22.
- For large i , the size of \tilde{Y}_i becomes small enough to make tabulation of $(\tilde{Y}_i)^i$ an attractive option. Again, this is especially relevant in the context of a table-based range reduction: there is already a table of coefficients, and it is possible to read in parallel the coefficients from one table and $(\tilde{Y}_i)^i$ from another.
- All the additions may be very efficiently merged in a compressor tree (as per Chap. 7). Actually, the whole polynomial evaluator may be designed as a single bit heap, receiving the partial products of

$$C_i Y^i = \left(\sum_j 2^j c_{i,j} \right) \left(\sum_k 2^k y_k \right)^i.$$

In this case, this expression is developed and simplified at the bit level, and the bits of weight lower than a certain threshold are discarded (the threshold being chosen to ensure that the sum of the discarded weighted bits is smaller than 2^{ℓ_R}). More conservatively, an intermediate power term $P_i = [(\tilde{Y}_i)^i]$ may be explicitly formed, either read from a table of pre-computed values or computed by an ad hoc powering unit, and then multiplied by C_i . Some previous works use a separate compressor tree for each product and throw into the bit heap a carry-save product, i.e., the result of the compression before the final addition.

Figure 18.20 illustrates a generic parallel evaluation architecture, irrespective of the actual implementation of the $[C_i x^i]$ boxes discussed above. Some of these options are depicted in Fig. 18.21. Which of these choices is the best is still, to our knowledge, an open question. The answer is probably quite dependent on the technology, the function, and the precision ℓ_R .

There are other minor optimizations to consider, for instance, the fact that the function x^i has the parity of i (it is odd if i is odd and even if i is even). This entails that the computation of $[C_i Y^i]$ for an even i can ignore the sign bit of Y . For odd i , it is possible to use XOR-based negation, as already used in Chap. 17 for the multipartite method (see Fig. 17.11).

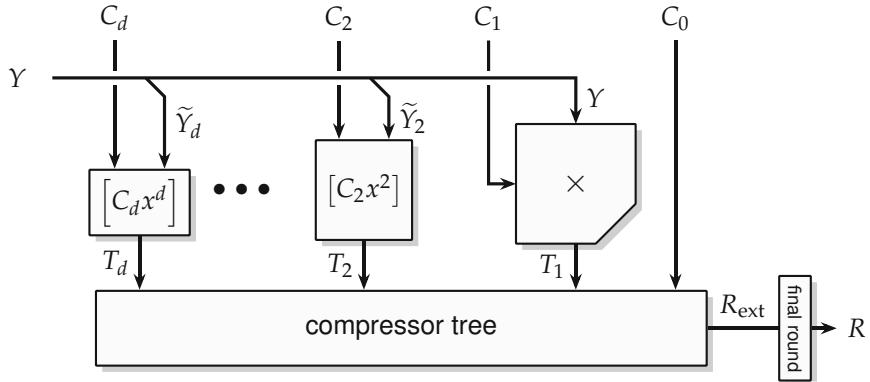


Fig. 18.20 Generic parallel polynomial evaluation architecture.

All in all, the literature only uses parallel polynomial evaluation for degrees up to 3 so far. Here is a selection of relevant works in chronological order. They all target 24-bit precision for single-precision floating point, and they all use a uniform piecewise domain decomposition:

- [PBM01] uses degree-2 architectures with a truncated squarer and a merged compressor tree, for the faithful evaluation of X^p for a floating-point X and $p \in \{-3, -2, -1, -1/2, 1/2\}$. This architecture is presented in detail for these functions but it is actually more generic.
- [OS05] uses a very similar degree-2 architecture whose parallel evaluator is designed to accommodate the functions $1/X$, $1/\sqrt{X}$, 2^X , $\log_2 X$, and sine/cosine. The architecture is not faithful (last-bit accurate) for all functions.
- The HOTBM method [DD05] is a generic approximation technique for arbitrary degree. For each $[C_i Y^i]$, a power term $P_i = [Y^i]$ is first computed, either by tabulation or by a dedicated powering unit. The word P_i is then decomposed into subwords $P_{k,i}$, and the same optimizations that have been introduced for the multipartite method in Chap. 17 are used for the products $C_i P_{k,i}$. In particular, C_i is actually $C_i(A)$, as it comes from a table indexed by A , the interval index (see Fig. 18.10). The products $C_i(A) P_{k,i}$, for the lower $P_{k,i}$, may be tabulated directly out of the bits of $P_{k,i}$ and a subset of leading bits of A , the interval index. This works thanks to the coefficient smoothness illustrated by Figs. 18.12, 18.13, and 18.14: this truncation of A entails the use of a neighboring coefficient $C_{i,\lfloor A \rfloor}$ that is an approximation of $C_{i,A}$. The error corresponding to this approximation must be accounted for in the error analysis [DD05].
- [Che15] uses a parallel degree-3 parallel architecture where the polynomials are the Taylor ones. Here, the Y^3 term is tabulated, while the Y^2 term uses a truncated squarer.

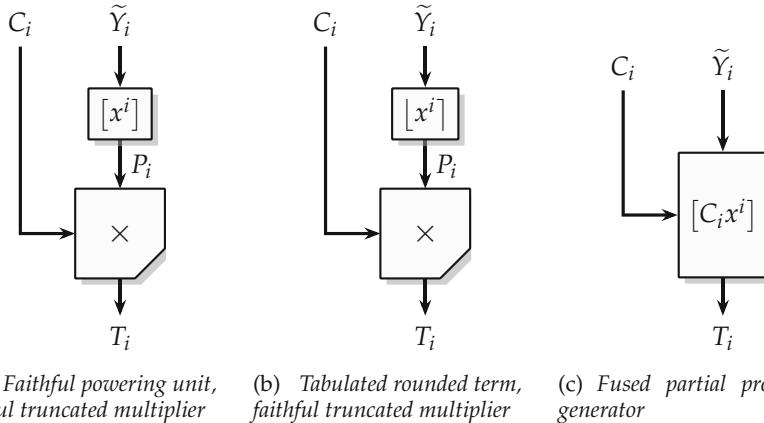


Fig. 18.21 Possible realizations of $[C_i x^i]$. For each option, there are at least three variants: T_i may be output as a set of weighted bits, or partially compressed in carry-save form, or fully compressed.

- [De +17] uses ILP to determine the optimal parameters of a degree-1 or degree-2 parallel architecture with truncated multiplier and squarer. This approach enables correct rounding of the function, albeit at a cost, whereas all other approaches can only achieve last-bit accuracy. It will be presented in more details in Sect. 18.4.4.

To the best of our knowledge, among all these works, only the HOTBM technique centers the reduced argument on its subinterval as suggested in Sect. 18.2.1.2. No method exploits the coefficient smoothness to compress the tables as suggested by Hsiao [Hsi+15], although the HOTBM method somehow exploits this property in a different way.

The error analysis for a parallel evaluator is typically simpler than for the Horner one, because there is no recurrence. As it is strongly dependent on the architectural choices made for each $[C_i Y^i]$, it is left as an exercise for the reader. The main parameter controlling the evaluation error is the LSB of the bit heap.

18.3.3 Other Polynomial Evaluation Techniques

Between the classic Horner scheme $p(x) = C_0 + x(C_1 + x(C_2 + \dots + xC_d) \dots)$ and the fully developed form $p(x) = C_0 + C_1 x + \dots + C_d x^d$, there are intermediate schemes that can be explored. For large degrees, the polynomial may be decomposed into an odd and an even part: $p(x) = p_e(x^2) + x \times p_o(x^2)$. The two sub-polynomials $p_e(x)$ and $p_o(x)$ may then be evaluated in parallel, so this scheme has a shorter latency than Horner, at the expense

of the precomputation of x^2 and a slightly degraded accuracy. The Estrin scheme is a recursive application of this idea [Mul16, §3.8.2]. It is useful only for large degrees, which probably explains why it has not been used in hardware to the authors' knowledge.

A polynomial with a degree of at least 3 may also be refactored to reduce the number of multiplications below the d multiplications of Horner scheme [Mul16, §3.8]. This family of techniques (which dates back to the 1960s) is well introduced in the PhD thesis of Revy [Rev09, §5.2] who considered them (then discarded them) in a software fixed-point context. The difficulty with such refactoring is that contrary to Estrin or Horner, it changes the coefficients, which adds an error term and possibly requires more accurate coefficients to start with. It is unclear if such refactoring is compatible with generic range reduction techniques.

However, a related idea has proven effective: the square rich method [XFM14] generalizes the simple refactoring of $c_0 + x(c_1 + xc_2)$ into

$$c_2 \left((x + m)^2 + n \right) \quad \text{with} \quad m = \frac{c_1}{2c_2} \quad \text{and} \quad n = \frac{c_0}{c_2} - \frac{c_1^2}{4c_2^2} \quad (18.31)$$

where one of the multiplications has been converted to a (cheaper) square. Based on operation count, this improves both area and latency compared to Horner. It is possible to take the additional error term into account [XFM14]. However, both m and n may now be much larger (in bits) than c_1 and c_2 . In particular, after a uniform piecewise argument reduction, we have $c_2 \ll c_1 \sim x$ (see, for example, Fig. 18.11) therefore $m \gg x$: in this case, we may replace a multiplier with a larger squarer. Still, improvements have been reported [XFM14] for degrees up to 6.

18.4 Putting It All Together: Generation of Polynomial Approximators

We now reconcile the polynomial approximation of Sect. 18.1, the range reduction techniques of Sect. 18.2, and the polynomial evaluation of Sect. 18.3 into software that generates polynomial approximators computing just right. The glue is an error analysis presented in Sect. 18.4.1. We then succinctly present two straightforward generators, one without range reduction in Sect. 18.4.2 and one with a uniform piecewise range reduction in Sect. 18.4.3. We then overview in Sect. 18.4.4 a recent technique that jointly optimizes the approximation and the evaluation steps in a single ILP problem.

18.4.1 Overall Error Analysis and Error Budget

The overall error of the operator is defined, as in previous chapters, as

$$\delta_{\text{total}}(X) = R(X) - f(X). \quad (18.32)$$

As introduced in Chap. 3, p. 65, the hardware must first compute an intermediate result R_{ext} with a precision slightly larger than the target precision 2^{ℓ_R} . The rounding of R_{ext} to the output format alone entails an error $\delta_{\text{final round}}$ that is bounded by 2^{ℓ_R-1} . In both Figs. 18.19 and 18.20, this rounding is actually for free: provided a half-ulp has been pre-added to C_0 , truncation of R_{ext} to LSB ℓ_R implements the rounding to the nearest of R_{ext} . This operation is nevertheless represented by the [final round] blocks in Figs. 18.19 and 18.20.

All the range reductions by interval splitting considered in Sect. 18.2 are exact: (18.3), for instance, introduces no error term, and in (18.4) or (18.6), we have $f_A(Y) = f(X)$. For this reason, $R_{\text{ext}}(X) = R_{\text{ext}}(A, Y)$ is simply denoted as R_{ext} .

We further define, as in Sect. 18.3, $\delta_{\text{eval}}(Y) = R_{\text{ext}} - P_A(Y)$ as the error that captures the accumulation of all the rounding errors in the architecture.

Equation (18.32) therefore becomes

$$\delta_{\text{total}}(X) = \underbrace{R - R_{\text{ext}}}_{=\delta_{\text{final round}}(X)} + \underbrace{R_{\text{ext}} - p_A(Y)}_{=\delta_{\text{eval}}(X)} + \underbrace{p_A(Y) - f_A(Y)}_{=\delta_{\text{approx}}(X)}. \quad (18.33)$$

Last-bit accuracy means that this error should be smaller than the ulp of the output:

$$\bar{\delta}_{\text{total}} < 2^{\ell_R}, \quad (18.34)$$

which by triangle inequality on (18.33) becomes the following constraint:

$$\bar{\delta}_{\text{final round}} + \bar{\delta}_{\text{eval}} + \bar{\delta}_{\text{approx}} < 2^{\ell_R} . \quad (18.35)$$

The general recipe for computing just right is to balance the sources of error. Therefore, the typical a priori distribution of the error budget is

- 2^{ℓ_R-1} for the final rounding,
- 2^{ℓ_R-2} for the approximation error,
- 2^{ℓ_R-2} for the evaluation error.

A typical algorithm is to use Algorithm 18.1 with $\bar{\delta}_{\text{approx}}^{\text{target}} = 2^{\ell_R-2}$. Algorithm 18.1 then reports the actual bound $\bar{\delta}_{\text{approx}} < \bar{\delta}_{\text{approx}}^{\text{target}}$. This implies that the actual evaluation error budget is

$$\bar{\delta}_{\text{eval}}^{\text{target}} = 2^{\ell_R-1} - \bar{\delta}_{\text{approx}} . \quad (18.36)$$

This is the value passed to the software that must dimension the hardware, e.g., Algorithm 18.2.

18.4.2 A Basic Polynomial Approximator Without Range Reductions

Figure 18.22 shows the execution flow of a basic polynomial approximator without any range reduction, as it is structured in the FloPoCo software. There is little choice in this case: to achieve last-bit accuracy to a given format, a minimal degree is required. The first step of the algorithm (1 in Fig. 18.22) is to invoke `BasicPolyApprox` to determine this degree d (using Sollya's `guessdegree()` command). `BasicPolyApprox` then determines an approximation polynomial p and reports the approximation error bound $\bar{\delta}_{\text{approx}}$. The generator then computes the error budget $\bar{\delta}_{\text{eval}}^{\text{target}}$ remaining for the evaluation hardware using (18.36) and passes it to the actual hardware generator, here a Horner evaluator that implements Algorithm 18.2 (2 in Fig. 18.22).

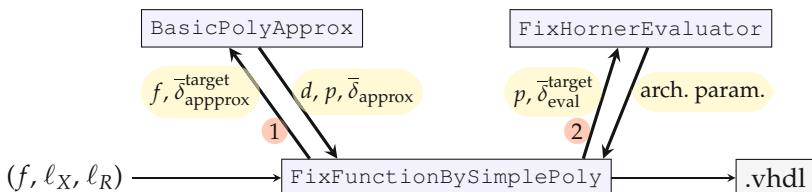


Fig. 18.22 Execution flow of a basic polynomial approximator .

Hands on: Polynomial Approximators Without Range Reduction

The main point of this FloPoCo operator is to experiment with polynomial approximation. For instance, the following command produces a polynomial evaluator for the exponential function on $[-1, 1]$:

```
flopoco FixFunctionBySimplePoly plainVHDL=true \
f="exp(x)" signedIn=true lsbIn=-24 lsbOut=-24
```

The `plainVHDL` option uses plain `*` and `+` for the additions instead of using the composite `FixMultAdd` operator. This makes the VHDL much easier to read. The two steps of Fig. 18.22 may be observed by adding the `verbose=2` option.

Observe how degree evolves when changing the values of the `lsbIn` and `lsbOut` parameters. Observe how the coefficients are close to the Taylor ones.

Now, we may reduce the interval and move it around using simple changes of variables, e.g.,

```
flopoco FixFunctionBySimplePoly plainVHDL=true \
f="exp(x/16+3/16)" signedIn=true lsbIn=-24 lsbOut=-24
```

The following command is an example where more argument reduction is needed: as the function is odd, half the coefficients are zeroes. The VHDL works, but is not optimal:

```
flopoco FixFunctionBySimplePoly plainVHDL=true \
f="(1-lb-24)*sin(pi/2*x)" \
signedIn=true lsbIn=-24 lsbOut=-24
```

18.4.3 A Polynomial Approximator with Uniform Piecewise Range Reduction

In the case of a range reduction, the algorithm is similar, as Fig. 18.23 shows.

First, `UniformPiecewisePolyApprox` is invoked. It itself invokes `BasicPolyApprox` for each subinterval. The number of subintervals 2^α is determined by `UniformPiecewisePolyApprox` simply by trial and error with increasing values of α : for each value of α , `UniformPiecewisePolyApprox` attempts to find approximation polynomials of degree d accurate to $\bar{\delta}_{\text{approx}}^{\text{target}} = 2^{\ell_R - 2}$ on all subinterval. A simple trick ensures quick rejections of values of α which are too small: for each value of α , `UniformPiecewisePolyApprox` first tries the two extremal intervals ($A = 0$ and $A = 2^\alpha - 1$) as the approximation error is likely to be maximal on one of these.

Once α is found, it would be possible to run Algorithm 18.1 on each subinterval. As already mentioned, a slightly better alternative is to have the loop on A inside the loop of Algorithm 18.1 determining ℓ_p : it ensures the same ℓ_p on all the intervals. Indeed, on some intervals, a smaller ℓ_p could be found, but it would result in zeroes at the LSB in the coefficient table. It is better to exploit these bits to store more accurate coefficients, which will allow for a smaller approximation error, hence a larger budget for the evaluation error.

Once all the approximation polynomials have been found, each with its approximation error, the evaluation error budgets are computed using (18.36) and passed to `FixHornerEvaluator` which runs Algorithm 18.2 on each interval, computing in the process all the remaining architectural parameters.

The interested reader is invited to read the source code of FloPoCo for the details which are omitted here.

Hands on: Polynomial Approximators with Uniform Piecewise Range Reduction

The following command produces a polynomial evaluator for the logarithm function between 1 and 2:

```
flopoco FixFunctionByPiecewisePoly plainVHDL=true \
f="log(1+x)" signedIn=false lsbIn=-24 lsbOut=-24 d=3
```

What is the value of α ? What is the table size? Observe how the table size and its aspect ratio evolve when changing the values of the degree d : try $d=2$, then $d=1$.

Compare the synthesis results, in your favorite environment, of `FixFunctionByMultipartiteTable` and `FixFunctionByPiecewisePoly` when $d=1$.

FloPoCo will report all the polynomials if you increase the verbosity level to `verbose=2`.

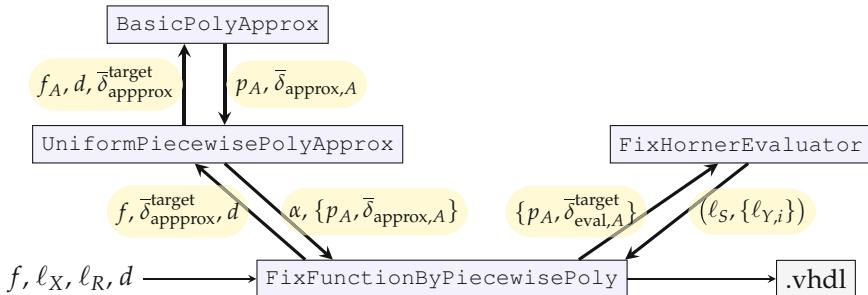


Fig. 18.23 Execution flow of a complete generator using uniform piecewise range reduction `FixFunctionByPiecewisePoly`.

18.4.4 Combined Approximation and Rounding Optimization Using integer linear programming (ILP)

The previous algorithm scales to precisions beyond 64 bits and arbitrary large degrees. For smaller precisions (up to 24 bits), a recent work [De +17] showed that both approximation and evaluation can be merged in a single ILP optimization problem. This approach even enables rounding to the nearest at a much lower cost than the worst-case error analysis of Sect. 18.4.1.

The basic idea of this method is quite simple. Considering that the main variables of the problem are simply the bits of the coefficients C_i , it consists in writing as many constraints as we have possible input values Y .

Let us note $c_{i,j}$ the bit of position j of each coefficient C_i or in other terms

$$C_i = \sum_{j=j_{\min,i}}^{j_{\max,i}} 2^j c_{i,j} \quad (18.37)$$

where $j_{\min,i}$ and $j_{\max,i}$ are bounds that can be determined beforehand, e.g., from a traditional polynomial approximation. Typically, $j_{\min,i}$ should be close to the value of ℓ_p determined by Algorithm 18.1.

The value computed by the architecture on input Y , assuming no rounding error, is then

$$\begin{aligned} R_{\text{ext}}(Y) &= \sum_{i=0}^d C_i Y^i \\ &= \sum_{i=0}^d \sum_{j=j_{\min,i}}^{j_{\max,i}} 2^j Y^i c_{i,j} . \end{aligned} \quad (18.38)$$

The constraint of correct rounding is (assuming the rounding bit is merged into C_0) that the truncation of R_{ext} is equal to the correct rounding of $f(X)$, which can be written

$$\lfloor f(X) \rfloor_{\ell_R} \leq R_{\text{ext}}(Y) < \lfloor f(X) \rfloor_{\ell_R} + 2^{\ell_R} . \quad (18.39)$$

A similar constraint can trivially be written for 1-ulp accuracy.

Combining (18.39) with (18.38), we have for each value of Y two linear constraints on the $c_{i,j}$. They can easily be transformed into integer linear constraints by multiplying by a suitable power of two: this defines an ILP problem.

There is an independent ILP problem for each subinterval, so we have 2^A ILP problems to solve, each with $2 \times 2^{w_X - \alpha}$ constraints. This shouldn't scale well, but with state-of-the-art solvers, the authors of [De +17] have applied it to precisions up to 24 bits with $\alpha = 7$.

Now, the beauty of this approach is that truncated products are managed equally well. Indeed, since the $P_i = Y^i$ are constants in the ILP problem, each occurrence of Y^i can be replaced with an arbitrarily truncated version in (18.38). For instance, the truncated multiplication $C_i P_i$, keeping only the partial products of binary position larger than ℓ , can be described as

$$\widetilde{C_i P_i} = \sum_{j=j_{\min,i}}^{j_{\max,i}} 2^j c_{i,j} \lfloor P_i \rfloor_{\ell-j} = \sum_{j=j_{\min,i}}^{j_{\max,i}} 2^j \underbrace{\lfloor P_i \rfloor_{\ell-j}}_{=K_{i,j}} c_{i,j} .$$

Further, any hardware approximation to Y^i can be modeled, provided it is faithfully reported in the ILP model. This is possible since we have one

constraint per value of Y : each of these constraints may use exactly the value of P_i output by the corresponding operator for this value of Y .

All that remains is to define an objective function. The objective here is to force to zero in priority the lower bits of C_i in order to minimize their size. This can be achieved by a bit of ILP engineering. Let us define binary variables $\mu_{i,k}$ such that

$$\mu_{i,k} = 1 \quad \text{iff} \quad c_{i,j} = 0 \quad \forall j \in \{j_{\min,i}, \dots, j_{\min,i} + k\} \quad (18.40)$$

This definition can be expressed recursively as linear inequalities. The objective function is then to maximize $\sum_{i,k} \mu_{i,k}$. Different weight can be given

to the bits of coefficients of different degree to reflect their hardware cost. However, the authors of [De +17] argue that it is better to perform the optimization in two passes, first optimizing the $C_1(A)$ and then the $C_2(A)$. The reason is similar to that already evoked in Sect. 18.4.3: the first pass will determine a $j_{\min,1}$ that will be the worst-case (i.e., smallest) value over all the intervals. For many intervals, it will be smaller than $j_{\min,1}$ determined on this interval, and this will give more room for the optimization of $j_{\min,2}$ during the second pass.

The interested reader is referred to the original work [De +17] for all the details omitted here. The technique is described for degree 1 and degree 2 only, but it is probably more general, although higher degrees do not help containing the size of the ILP problems since a higher degree entails a smaller α . The reported results (in terms of coefficient sizes) of this method are constantly, if marginally, better than those of the generator presented in Sect. 18.4.3: a few bits are shaved from each coefficient.

18.5 Floating-Point Architectures Based on Fixed-Point Polynomials

To conclude this chapter, let us stress that its focus on fixed point does not reduce the usefulness of the methods surveyed here. For instance, efficient hardware architectures for floating-point functions [PBM01; OS05; LP13; Che15; JP20] often rely on a fixed-point core, as already mentioned in Sect. 18.3.2. Chapter 22 will provide a detailed example of using a fixed-point polynomial to build a floating-point elementary function.

When the input X is a floating-point number, its exponent is a crude approximation of $\log_2(X)$ and therefore provides a logarithmic segmentation almost for free [JP20]. This approximation can be refined by also considering a few of the leading bits of the significand, input into a very small look-up table. This has been used for the probit function, i.e., the inverse cumulative distribution function of the Gaussian distribution [JP20].

Another significant example is an algorithm constructing a universal floating-point approximator [Tho15]. It inputs an arbitrary function and also a degree d which may be used as a knob controlling the memory/arithmetic trade-off just as in the uniform piecewise approximator of Sect. 18.4.3. The algorithm first splits the floating-point input domain into sub-domains such that neither the input nor its image change exponents within a subdomain: this effectively reduces the problem to a fixed-point one. Then these sub-domains are further split, if needed, until on each sub-domain the function is monotonic and can be faithfully approximated by a polynomial of degree d . A table stores, for each sub-domain, the coefficients of the polynomial but also the exponent of the output. In this work, the segmentation is arbitrary (other segmentation schemes make much less sense in a floating-point context), and the segment index is found by dichotomy. Finally, a fixed-point approximator able to cope with all these polynomials is constructed, here using a Horner scheme.

References

- [BC07] Nicolas Brisebarre and Sylvain Chevillard. “Efficient Polynomial L^∞ -approximations”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2007, pp. 169–176 (cit. on p. [537](#)).
- [BM04] Nicolas Brisebarre and Jean-Michel Muller. “Functions Approximable by E-Fractions”. In: *Asilomar Conference on Signals, Circuits and Systems*. Vol. 2. IEEE, 2004, pp. 1341–1344 (cit. on p. [540](#)).
- [Che+07] Ray C.C. Cheung, Dong-U. Lee, Wayne Luk, and John D. Villasenor. “Hardware Generation of Arbitrary Random Number Distributions From Uniform Distributions Via the Inversion Method”. In: *IEEE Transactions on VLSI Systems* 8.15 (2007) (cit. on p. [554](#)).
- [Che+11] Sylvain Chevillard, John Harrison, Mioara Joldes, and Christoph Lauter. “Efficient and accurate computation of upper bounds of approximation errors”. In: *Theoretical Computer Science* 412.16 (2011), pp. 1523–1543 (cit. on pp. [534](#), [538](#)).
- [Che15] Chichyang Chen. “High-order Taylor series approximation for efficient computation of elementary functions”. In: *IET Computers & Digital Techniques* 9.6 (2015), pp. 328–335 (cit. on pp. [542](#), [560](#), [568](#)).
- [CJL10] Sylvain Chevillard, Mioara Joldes, and Christoph Lauter. “Sollya: An Environment for the Development of Numerical Codes”. In: *Mathematical Software - ICMS 2010*. Vol. 6327. Lecture Notes in Computer Science. Springer, 2010, pp. 28–31 (cit. on p. [537](#)).
- [CK12] Dongdong Chen and Seok-Bum Ko. “A dynamic non-uniform segmentation method for first-order polynomial function evalua-

- tion". In: *Microprocessors and Microsystems* 36 (2012), pp. 324–332 (cit. on p. 552).
- [DD05] Jérémie Detrey and Florent de Dinechin. "Table-based polynomials for fast hardware function evaluation". In: *International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE, 2005, pp. 328–333 (cit. on p. 560).
- [De +17] Davide De Caro, Ettore Napoli, Darjn Esposito, Gerardo Castellano, Nicola Petra, and Antonio GM Strollo. "Minimizing Coefficients Wordlength for Piecewise-Polynomial Hardware Function Evaluation With Exact or Faithful Rounding". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 64.5 (2017), pp. 1187–1200 (cit. on pp. 542, 561, 566, 567, 568).
- [Din+10] Florent de Dinechin, Mioara Joldes, Bogdan Pasca, and Guillaume Revy. "Multiplicative square root algorithms for FP-GAs". In: *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2010, pp. 574–577 (cit. on p. 553).
- [DJP10] Florent de Dinechin, Mioara Joldes, and Bogdan Pasca. "Automatic generation of polynomial-based hardware architectures for function evaluation". In: *Application-specific Systems, Architectures and Processors*. IEEE, 2010 (cit. on pp. 529, 541, 542).
- [DT05] Florent de Dinechin and Arnaud Tisserand. "Multipartite Table Methods". In: *IEEE Transactions on Computers* 54.3 (2005), pp. 319–330 (cit. on p. 548).
- [Hsi+15] Shen-Fu Hsiao, Po-Han Wu, Chia-Sheng Wen, and Pramod Kumar Meher. "Table Size Reduction Methods for Faithfully Rounded Lookup-Table-Based Multiplierless Function Evaluation". In: *Transactions on Circuits and Systems II* 62.5 (2015), pp. 466–470 (cit. on pp. 544, 561).
- [JP20] Mioara Joldes and Bogdan Pasca. "Efficient Floating-Point Implementation of the Probit Function on FPGAs". In: *International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE, 2020, pp. 173–180 (cit. on pp. 554, 568).
- [KL14] Olga Kupriianova and Christoph Quirin Lauter. "A Domain Splitting Algorithm for the Mathematical Functions Code Generator". In: *Asilomar Conference on Signals, Circuits and Systems*. IEEE, 2014 (cit. on p. 552).
- [Lee+03] Dong-U Lee, Wayne Luk, John Villasenor, and Peter Cheung. "Hierarchical Segmentation Schemes for Function Evaluation". In: *International Conference on Field-Programmable Technology (FPT)*. IEEE, 2003 (cit. on p. 551).
- [Lee+05] Dong-U Lee, Altaf A. Gaffar, Oskar Mencer, and Wayne Luk. "Optimizing Hardware Function Evaluation". In: *IEEE Transactions on Computers* 54.12 (2005), pp. 1520–1531 (cit. on pp. 529, 536, 541, 542).

- [Lee+08] Dong-U Lee, Ray Cheung, Wayne Luk, and John Villasenor. "Hardware implementation trade-offs of polynomial approximations and interpolations". In: *IEEE Transactions on computers* 57.5 (2008), pp. 686–701 (cit. on pp. 547, 548).
- [Lee+09] Dong-U Lee, Peter Cheung, Wayne Luk, and John Villasenor. "Hierarchical Segmentation Schemes for Function Evaluation". In: *IEEE Transactions on VLSI Systems* 17.1 (2009) (cit. on pp. 529, 530, 536, 542, 551).
- [LP13] Martin Langhammer and Bogdan Pasca. "Elementary Function Implementation With Optimized Sub-Range Polynomial Evaluation". In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2013, pp. 202–205 (cit. on p. 568).
- [Mul16] Jean-Michel Muller. *Elementary Functions, Algorithms and Implementation*. 3rd ed. Birkhäuser Boston, 2016 (cit. on pp. 529, 530, 531, 533, 540, 562).
- [OS05] Stuart F. Oberman and Ming Y. Siu. "A High-Performance Area-Efficient Multifunction Interpolator". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2005 (cit. on pp. 542, 560, 568).
- [PBM01] José-Alejandro Piñeiro, Javier D. Bruguera, and Jean-Michel Muller. "Faithful Powering Computation Using Table Look-Up and a Fused Accumulation Tree". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2001, pp. 40–47 (cit. on pp. 542, 560, 568).
- [Rem34] Evgeniy Remez. "Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation". In: *Comptes Rendus de l'Académie des Sciences, Paris* 198 (1934) (cit. on p. 533).
- [Rev09] Guillaume Revy. "Implementation of binary floating-point arithmetic on embedded integer processors : Polynomial evaluation-based algorithms and certified code generation." PhD thesis. École Normale Supérieure de Lyon, 2009 (cit. on p. 562).
- [SDP11] Antonio G.M. Strollo, Davide De Caro, and Nicola Petra. "Elementary Functions Hardware Implementation Using Constrained Piecewise-Polynomial Approximations". In: *IEEE Transactions on Computers* 60.3 (2011), pp. 418–432 (cit. on p. 543).
- [SM04] Tsutomu Sasao and Munehiro Matsuura. "A Method to Decompose Multiple-Output Logic Functions". In: *Design Automation Conference (DAC)*. ACM/IEEE, 2004, pp. 428–433 (cit. on p. 552).
- [SNB05] Tsutomu Sasao, Shinobu Nagayama, and Jon T. Butler. "Programmable Numerical Function Generators: Architectures and Synthesis Method". In: *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2005, pp. 118–123 (cit. on p. 552).

- [Tho15] David B. Thomas. “A general-purpose method for faithfully rounded floating-point function approximation in FPGAs”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2015 (cit. on pp. [530](#), [552](#), [569](#)).
- [XFM14] Simin Xu, Suhaib A. Fahmy, and Ian V. McLoughlin. “SquareRich Fixed Point Polynomial Evaluation on FPGAs”. In: *International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM, 2014, pp. 99–108 (cit. on p. [562](#)).



19

CHAPTER 19

Digit Recurrence for Algebraic Functions

An algebraic function is defined as the solution to a multivariate polynomial equation. This definition can often be combined with the equation of a radix- β representation to obtain a digit-recurrence algorithm evaluating the function. This is a generalization of a technique described in detail in Chap. 9 for division (another algebraic function). It is first demonstrated on square and cube roots, then on 2D and 3D Euclidean norms, and finally on the evaluation of an arbitrary rational function.

This chapter generalizes some of the ideas introduced for division to many algebraic functions. It focuses on digit-recurrence algorithms, which compute the result one digit at a time. Here, the notion of digit depends on the radix β used, and the algorithms will be formulated in a generic way that works for any β . As for division in Chap. 9, different values of β (usually small powers of two) will provide as many trade-offs between the number of iterations and the complexity of an iteration. It may also be more comfortable to some readers if it allows them to think in decimal by considering $\beta = 10$.

In general, a radix- β description of the problem will provide a generic but incomplete algorithm, and the details will be refined for each function to be evaluated.

Section 19.1 addresses the square root function, Sect. 19.2 addresses the cube root function, Sect. 19.3 addresses Euclidean norms $\sqrt{X^2 + Y^2}$ or $\sqrt{X^2 + Y^2 + Z^2}$, and finally Sect. 19.4 introduces the E-method which can be used to evaluate arbitrary rational functions. Table 19.1 provides some common notations consistently used in this chapter, for future reference.

Table 19.1 Common notations used in this chapter.

Symbol	Meaning
β	The base or radix
α	When β is a power of two, $\beta = 2^\alpha$
γ	Digit set parameter: digits belong to $\{-\gamma, \dots, 0, \dots, \gamma\}$
ρ	The redundancy factor, $\rho = \frac{\gamma}{\beta-1}$
d_j	A radix- β digit of the function being evaluated
S_i	Increasingly accurate approximations to the function being evaluated
W_i	Scaled remainder at iteration i

19.1 Digit-Recurrence Square Root

19.1.1 Range Reduction for Floating-Point and Fixed-Point Binary

The square root is defined over positive real numbers. For binary inputs, its range reduction is based on the following identity:

$$\sqrt{2^{2p}x} = 2^p\sqrt{x}. \quad (19.1)$$

Let us first address the square root of a positive floating-point normalized number $X = 2^E \cdot 1.F$, with E an integer exponent and $1.F \in [1, 2)$:

- If E is even, (19.1) can be used directly with $2p = E$ and $\sqrt{X} = 2^{E/2}\sqrt{1.F}$.
- If E is odd, X must be denormalized by one bit: $X = 2^{E-1} \cdot (2 \times 1.F)$. Then (19.1) can be used with $2p = E - 1$ and $\sqrt{X} = 2^{(E-1)/2}\sqrt{2 \times 1.F}$ where $(E-1)/2$ is an integer, hence a valid exponent, and $2 \times 1.F \in [2, 4)$.

In summary, the square root of a floating-point binary input may be computed as

$$\sqrt{2^E \cdot 1.F} = 2^{E'/2}\sqrt{M} \quad (19.2)$$

with

$$\begin{cases} E' = E, & M = 1.F \quad \text{if } E \text{ even} \\ E' = E - 1, & M = 2 \times 1.F \quad \text{if } E \text{ odd} \end{cases} \quad (19.3)$$

and $M \in [1, 4)$ as illustrated by Fig. 19.1.¹

If the input is provided as a fixed-point number with a large range, it may also be a good idea to first convert it to a floating-point format with an even exponent. The hardware needed for this is a normalizer (see Sect. 10.3) combining a leading zero counter and a shifter. After the floating-point square

¹ Some authors prefer to reduce to $M \in [\frac{1}{4}, 1)$ – it makes no difference in practice.

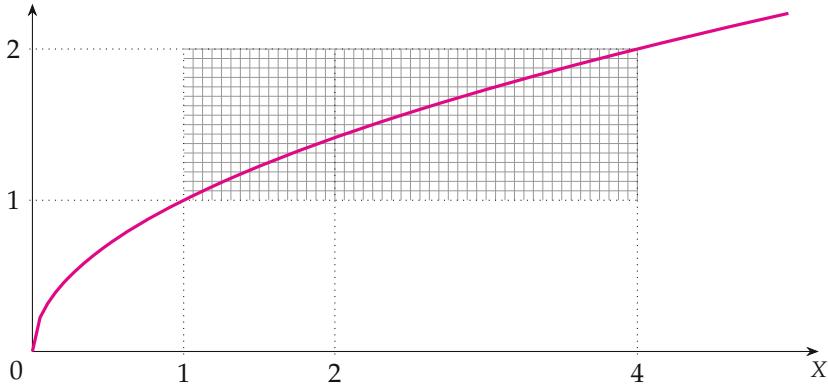


Fig. 19.1 Square root range reduction.

root, a second shifter is needed to convert the result back to fixed point. This conversion is particularly relevant when using an approximation method that exploits the derivability of the function, such as polynomial approximation: the square root is not derivable in 0, and therefore these methods do not work well near 0. Conversely, this range reduction is not needed if plain tabulation is used.

The remainder of this section focuses on the square root of a fixed-point number $M \in [1, 4)$, represented in the $\text{ufix}(1, \ell)$ format for some integer $\ell < 0$.

19.1.2 Generic Radix- β Formulation

The square root will be obtained digit by digit, most significant digit first. Iteration i will determine a new digit d_{-i} (in radix β), such that the number S_i formed by these digits

$$S_i = \sum_{j=0}^i \beta^{-j} d_{-j} \quad (19.4)$$

is an increasingly accurate approximation to \sqrt{M} . As $M \in [1, 4)$ ensures that $\sqrt{M} \in [1, 2)$, the digit d_0 is known: $d_0 = 1$ and therefore $S_0 = 1$.

Assume for now that we have such an iteration. A measure of the accuracy of S_i would be the error $S_i - \sqrt{M}$, but it is difficult to evaluate within the algorithm since \sqrt{M} is unknown. Instead, we capture the accuracy of S_i by defining the remainder $S_i^2 - M$, which is easy to compute exactly – this is where we exploit the fact that square root is an algebraic function.

Let us first consider an integer square root, analogue to Euclidean (integer) division. The square root of an integer N can be defined as an integer square root S and an integer remainder R (all positive) such that $N = S^2 + R$.

For unicity, we add the condition that $S^2 \leq N < (S + 1)^2$ which, by introducing $R = N - S^2$, can be rewritten $0 \leq R < 2S + 1$.

In the formulation above, the size of the interval of S is 1, which is the ulp of integers. Now, if we move to a more general fixed-point formulation, where iteration i computes a square root accurate to β^{-i} , then the ulp becomes β^{-i} , and the unicity constraint becomes $S_i^2 \leq M < (S_i + \beta^{-i})^2$ which, defining the remainder $R'_i = M - S_i^2$, can be rewritten $0 \leq R'_i < 2\beta^{-i}S_i + \beta^{-2i}$.

It is more convenient to multiply this inequality by β^i , thus defining a scaled remainder:

$$W_i = \beta^i(M - S_i^2) \quad (19.5)$$

so that the scale of the bound is invariant with each iteration:

$$0 \leq W_i < 2S_i + \beta^{-i} \quad (19.6)$$

As $R'_i = M - S_i^2 \geq 0$, this constraint corresponds to a rounding down of the square root. As for division, it is possible to recenter this constraint to target rounding to nearest instead, e.g.,

$$-S_i \leq W_i < S_i + \beta^{-i-1} \quad (19.7)$$

Contrary to division, this bound is not constant due to the β^{-i} term.

The remainder for iteration $i + 1$ can then be computed as

$$W_{i+1} = \beta^{i+1}(M - S_{i+1}^2) \quad (19.8)$$

$$= \beta^{i+1} \left(M - (S_i + \beta^{-i-1}d_{-i-1})^2 \right) \quad (19.9)$$

$$= \beta W_i - 2d_{-i-1}S_i - \beta^{-i-1}d_{-i-1}^2 \quad . \quad (19.10)$$

Similar to division, a digit-recurrence algorithm for square root consists of determining d_{-i-1} out of W_i and S_i in such a way that W_{i+1} will remain within the bounds (19.6) or (19.7) and then evaluating (19.10) to compute W_{i+1} . Meanwhile, S_{i+1} is obtained by appending d_{-i-1} to S_i according to (19.4).

Let us now see some practical examples.

19.1.3 Simple Binary Restoring Square Root

In binary ($\beta = 2$), d_{-i} is either 0 or 1. The simplest algorithm therefore consists in assuming $d_{-i} = 1$, attempting the subtraction (19.10), and testing the sign of the result: if it is negative, then the constraint (19.6) is violated,

and the proper digit to select was $d_{-i} = 0$. The complete algorithm is below (read T_{i+1} as “tentative next remainder”) and the corresponding architecture is given in Fig. 19.2:

$$d_0 = 1 \quad (19.11)$$

$$S_0 = 1. \quad (19.12)$$

$$W_0 = M - 1. \quad (19.13)$$

$$\forall 0 \leq i < n - 1 \quad T_{i+1} = 2W_i - (2S_i + 2^{-i-1}) \quad (19.14)$$

$$d_{-i-1} = \begin{cases} 1 & \text{if } T_{i+1} \geq 0 \\ 0 & \text{if } T_{i+1} < 0 \end{cases} \quad (19.15)$$

$$W_{i+1} = \begin{cases} T_{i+1} & \text{if } T_{i+1} \geq 0 \\ 2W_i & \text{if } T_{i+1} < 0 \end{cases} \quad (19.16)$$

$$S_{i+1} = S_i + 2^{-i-1}d_{-i-1} \quad (19.17)$$

There is no need to use the centered constraint: Rounding to the nearest for precision ℓ is easily obtained by computing one extra bit of accuracy, i.e., having $n = -\ell + 1$ and then using this extra bit d_{-n} as round bit. The square root rounded to nearest is then $S_{n-1} + 2^\ell d_{-n}$. This implements rounding to the nearest with ties to up, but ties typically do not happen (see Sect. 11.4.3, p. 356).

In practice, S_i is in the ufix(0, $-i$) format; therefore, (19.17) does not require addition hardware: it is a bit concatenation. The same holds for the subtrahend $2S_i + 2^{-i-1}$ in (19.14): $2S_i$ is in the ufix(1, $-i + 1$) format, therefore $2S_i + 2^{-i-1}$ is a ufix(1, $-i - 1$) binary number obtained by appending 01₂ to the right of $2S_i$.

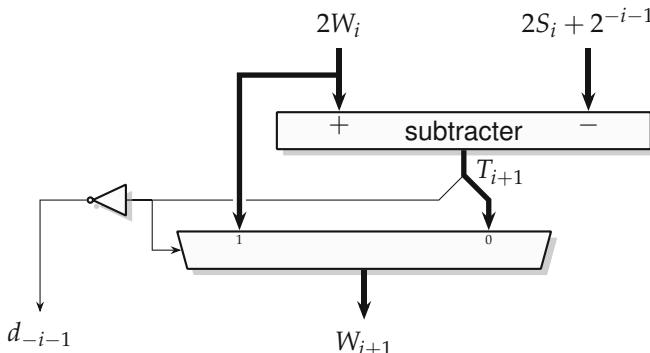


Fig. 19.2 Architecture of one step of a restoring square root.

The tentative next remainder T_{i+1} is therefore a $\text{sfix}(1, -i - 1)$ number. The corresponding effective subtraction can be limited to position $-i - 1$ at the LSB: the bits of W_{i+1} lower than this position are simply copied from $2W_i$ (and, by recurrence, from $W_0 = M - 1$). This is quite similar to the paper-and-pencil division, where each iteration consumes a new bit of the dividend.

If the recurrence is unrolled in space, iteration i requires a subtracter on only $i + 3$ bits, so the first iterations are faster than the last ones.

Hands on: A Restoring Floating-Point Square Root in FloPoCo

The following command produces a floating-point square root using the plain restoring algorithm above:

```
flopoco FPSqrt method=0 wE=8 wF=23
```

19.1.4 Binary Non-restoring Square Root

As for division, an interesting variant is to use T_i instead of W_i as the current remainder. We then want to replace (19.16) with a recursive definition of T_{i+1} as a function of T_i . This is derived by distinguishing the two cases:

- If in the previous iteration we had $T_i \geq 0$, meaning that $d_i = 1$, then we had $T_i = W_i$ (from (19.16) at iteration (i)). So we can replace W_i with T_i in (19.14), and we obtain

$$T_{i+1} = 2T_i - (2S_i + 2^{-i-1}) \quad . \quad (19.18)$$

- If in the previous iteration we had $T_i < 0$, meaning that $d_i = 0$, then to express W_i out of T_i , we need to “restore” what was subtracted from the tentative remainder: in this case, $W_i = T_i + 2S_{i-1} + 2^{-i}$. Therefore, we may rewrite (19.14) as

$$T_{i+1} = 2(T_i + 2S_{i-1} + 2^{-i}) - 2S_i - 2^{-i-1} \quad (19.19)$$

But in this case, $S_i = S_{i-1}$ since $d_i = 0$; hence,

$$T_{i+1} = 2T_i + (2S_i + 2^{-i+1} - 2^{-i-1}) \quad (19.20)$$

In this case, the value added to $2S_i$ is $2^{-i+1} - 2^{-i-1}$ and is equal to $3 \cdot 2^{-i-1}$ (it consists of two ones in binary).

The additions to $2S_i$ (of $3 \cdot 2^{-i-1}$ in (19.20) and of 2^{-i-1} in (19.18)) are in both cases simple bit concatenations, as $2S_i$ is in the $\text{ufix}(1, -i + 1)$ format.

The formula $2^{-i}\overline{d_i} + 2^{-i-1}$ captures the addends of both cases, and we obtain the non-restoring algorithm:

$$S_{-1} = 0 \quad (19.21)$$

$$T_0 = M - 1 \quad (19.22)$$

$$\forall 0 \leq i < n-1 \quad d_{-i} = \begin{cases} 1 & \text{if } T_i \geq 0 \\ 0 & \text{if } T_i < 0 \end{cases} \quad (19.23)$$

$$S_i = S_{i-1} + 2^{-i}d_{-i} \quad (19.24)$$

$$U_i = 2S_i + 2^{-i}\overline{d_{-i}} + 2^{-i-1} \quad (19.25)$$

$$T_{i+1} = \begin{cases} 2T_i + U_i & \text{if } d_{-i} = 0 \\ 2T_i - U_i & \text{if } d_{-i} = 1 \end{cases} \quad (19.26)$$

The initialization (19.22) is identical to the restoring case.

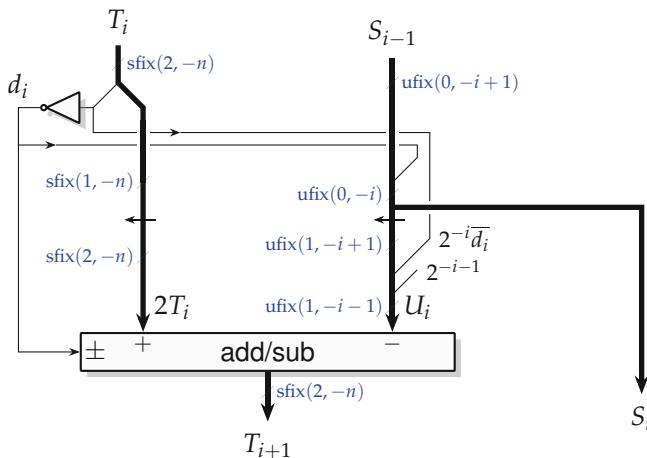


Fig. 19.3 Architecture of one step of a non-restoring square root.

As Fig. 19.3 shows, this algorithm is very hardware-friendly: (19.23) simply takes the Boolean negation of the sign bit of T_i ; (19.24) and (19.25) are simple bit concatenations, so the only hardware required is an inverter and an adder/subtractor for (19.26).

Besides, U_i is obviously in the ufix($1, -i - 1$) format. Therefore, as for the restoring algorithm, the actual addition/subtraction to perform in (19.26) only affects the bits to the left of position $-i - 1$. The remaining LSB bits are copied from $2T_i$ (hence, recursively, from the input M). For the sake of simplicity, this is not shown in Fig. 19.3.

The sign information, however, must be kept in the subtraction: the effective adder/subtractor computes bits with indices ranging from $-i - 1$ to 2 (i.e., $i + 4$ bits).

In the experiments by the authors on current FPGAs, this non-restoring formulation maps better to their LUT and fast carry structure, entailing lower area and critical path than the restoring one. It is therefore the current default in FloPoCo.

Hands on: A Non-restoring Floating-Point Square Root in FloPoCo

The following command produces a floating-point square root using the non-restoring algorithm above:

```
flopoco FPSqrt wE=8 wF=23
```

19.1.5 Exploiting Redundant Number Systems in the Square Root

Let us get back to the general radix- β formulation of the problem. For convenience, we copy here (19.10), the definition of the next partial remainder to be computed at iteration i :

$$W_{i+1} = \beta W_i - 2d_{-i-1}S_i - \beta^{-i-1}d_{-i-1}^2 . \quad (19.27)$$

Here also, S_i grows with each iteration, so the early iterations involve only a very short effective subtraction (digit positions from 0 to $-i - 1$). As iterations progress, the carry propagation gets longer, and the iteration delay is dominated by the carry propagation in (19.27). A way to avoid this is some form of carry-save computation of W_{i+1} . Unfortunately, it is not possible to decide the sign of a carry-save representation (hence, d_{-i-1}) without first converting it to standard representation, which again involves a carry propagation.

As for division, the solution is to rely on a redundant number representation. If the next digit is taken from a redundant digit set, it can be selected on the basis of an approximation of the partial remainder obtained by considering only a few of its leading digits. The carry propagation needed inside each iteration can then be limited to these few leading digits.

As for division, the idea is to replace the constraint on the scaled remainder (19.7) with a relaxed constraint. For this, we define the absolute error

$$\delta_i = S_i - \sqrt{M} . \quad (19.28)$$

Since $S_i = \sum_{j=0}^i \beta^{-j}d_{-j}$ from (19.4), we look for a bound of the form

$$|\delta_i| \leq \rho\beta^{-i} \quad (19.29)$$

when we use the redundant digit set $d_{-j} \in \{-\gamma, \dots, \gamma\}$. Imposing (19.29) on $\delta_{i+1} = S_{i+1} - \sqrt{M} = \delta_i + \beta^{-i-1}d_{-i-1}$ yields

$$|\delta_i + \beta^{-i-1}d_{-i-1}| \leq \rho\beta^{-i-1} \quad (19.30)$$

The worst positive error, $\delta_i = \rho\beta^{-i}$, can be compensated by the smallest possible value of d_{-i-1} , or $d_{-i-1} = -\gamma$. Then, (19.30) becomes $\rho\beta^{-i} - \gamma\beta^{-i-1} \leq \rho\beta^{-i-1}$ which simplifies in $\rho \leq \gamma/(\beta - 1)$. The computation on the worst positive case is symmetrical and yields the same result. We keep the largest possible value of ρ since it will provide the most freedom and therefore define the same redundancy factor ρ as for division:²

$$\rho = \frac{\gamma}{\beta - 1} \quad . \quad (19.31)$$

Now, the constraint (19.29) can be rewritten using (19.28) as $-\rho\beta^{-i} \leq S_i - \sqrt{M} \leq \rho\beta^{-i}$ or $S_i - \rho\beta^{-i} \leq \sqrt{M} \leq S_i + \rho\beta^{-i}$. The three members are always positive so we can take the square of both inequalities, yielding $S_i^2 + \rho^2\beta^{-2i} - 2\rho\beta^{-i}S_i \leq M \leq S_i^2 + \rho^2\beta^{-2i} + 2\rho\beta^{-i}S_i$. Subtracting S_i^2 and then multiplying by β^i give $\rho^2\beta^{-i} - 2\rho S_i \leq \beta^i(M - S_i^2) \leq \rho^2\beta^{-i} + 2\rho S_i$ where we recognize the scaled remainder $W_i = \beta^i(M - S_i^2)$ of (19.5). What we have here is the equivalent of the convergence constraint (19.29) but expressed as a constraint linking the current approximation S_i and the scaled remainder W_i :

$$-2\rho S_i + \rho^2\beta^{-i} \leq W_i \leq 2\rho S_i + \rho^2\beta^{-i} \quad . \quad (19.32)$$

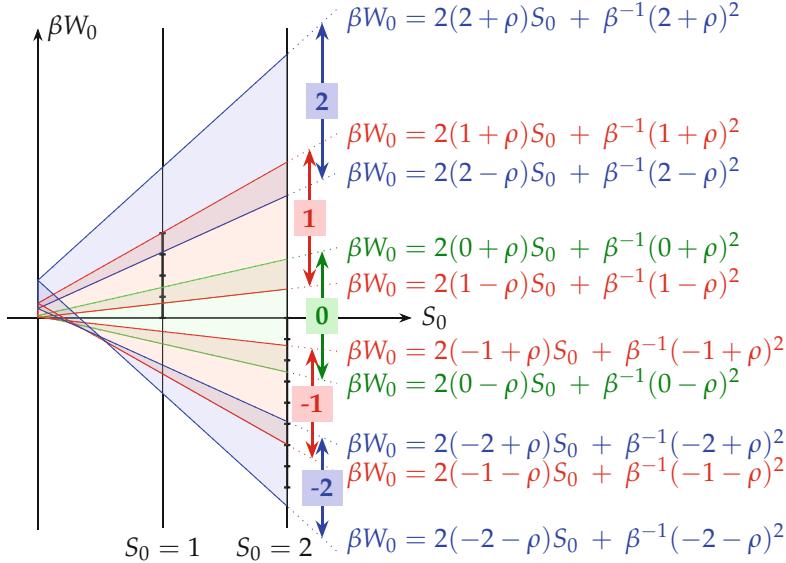
It remains to transform (19.32) into a constraint on the next digit d_{-i-1} that can be used to define a selection function. For this, we impose (19.32) on W_{i+1} defined by (19.10) as $W_{i+1} = \beta W_i - 2d_{-i-1}S_i - \beta^{-i-1}d_{-i-1}^2$. This yields $-2\rho S_{i+1} + \rho^2\beta^{-i-1} + 2d_{-i-1}S_i + \beta^{-i-1}d_{-i-1}^2 \leq \beta W_i \leq 2\rho S_{i+1} + \rho^2\beta^{-i-1} + 2d_{-i-1}S_i + \beta^{-i-1}d_{-i-1}^2$. After replacing S_{i+1} with $S_i + \beta^{-i-1}d_{-i-1}$, this constraint slightly simplifies as

$$2(d_{-i-1} - \rho)S_i + \beta^{-i-1}(d_{-i-1} - \rho)^2 \leq \beta W_i \quad (19.33)$$

$$\beta W_i \leq 2(d_{-i-1} + \rho)S_i + \beta^{-i-1}(d_{-i-1} + \rho)^2 \quad . \quad (19.34)$$

These constraints (19.33) and (19.34) define the freedom of the selection function that will determine d_{-i-1} at iteration i . For each iteration, it is

² Indeed the determination of ρ is independent on the function being computed, here \sqrt{M} : we have only used (19.4), which defines a radix- β number system, and the digit set $d_{-j} \in \{-\gamma, \dots, \gamma\}$. The redundancy factor ρ is a function of the number system only.



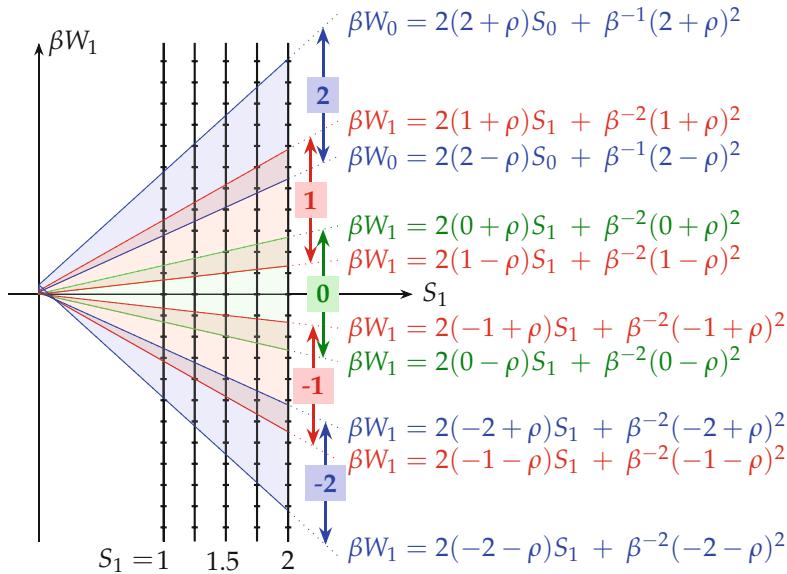


Fig. 19.5 P-D diagram at iteration $i = 1$ for the square root in radix 4 with $\gamma = 2$.

of d_0 , and in our case, there is not much choice; it will be $d_0 = 2$. A valid selection of d_0 is the following:

- If $M \in [1, 2)$, then $d_0 = 1 = S_0$; hence $W_0 = \beta^0(M - S_0^2) \in [0, 1)$.
- If $M \in [2, 4)$, then $d_0 = 2 = S_0$; hence, $W_0 = \beta^0(M - S_0^2) \in [-2, 0)$, and (19.32) is respected since $-2\rho S_0 + \rho^2 \beta^0 \approx -2.222$.

We therefore only have two possible values for S_0 to consider in the P-D diagram of iteration 0, represented in Fig. 19.4. For each value of S_0 , the domain of βW_0 is itself restricted. This diagram determines d_{-1} , which can take any value from -2 to 2. The computation of W_1 is by

$$W_1 = \beta W_0 - 2d_{-1}S_0 - \beta^{-1}d_{-1}^2, \quad (19.35)$$

and we may assume that the small subtraction is performed completely (with carry propagation). At iteration $i = 1$, we have the following possible values of S_1 to consider (see Fig. 19.4): 1.0, 1.1, 1.2, 2.2̄, 2.1̄, and 2.0 or in binary 01.00, 01.01, 01.10, 01.10, 01.11, and 10.00. These five values are plotted in Fig. 19.5, where it can be seen that the selection of digit d_{-1} can be made by considering the 4 bits of S_1 and a truncation to 5 bits of W_1 .

S_2 will then be coded on 6 bits, and for the following iterations, it becomes interesting to consider its truncation in the PD diagram.

19.1.5.2 Alternative Initialization by Tabulation

An alternative to the two first iterations described above is to look up S_2 in a table addressed by the leading bits of M and then compute W_2 directly as $W_2 = \beta^2(M - S_2^2)$. To reduce latency, S_2^2 can be looked up as well from the same table. And of course, this approach can be pushed to any $i_0 \geq 2$ at increasing table cost.

How many bits should be input to this table? To ensure convergence, the error $\delta_{i_0} = S_{i_0} - \sqrt{M}$ should be bounded by (19.29): $|\delta_{i_0}| \leq \rho\beta^{-i_0}$. The table values must be rounded to $\beta^{-i_0} = 2^{-\alpha i_0}$, for instance, the LSB of S_2 is $4^{-2} = 2^{-4}$. Let us call ℓ the bit position at which we truncate M before inputting it to the table, and let us define

$$\tilde{M} = \lfloor M \rfloor_\ell. \quad (19.36)$$

We could tabulate $\left\lfloor \sqrt{\tilde{M}} \right\rfloor_{-\alpha i_0}$, but the error can be reduced (for free) by taking the evaluation point at the center of the interval of values of M represented by \tilde{M} :

$$S_{i_0} = \left\lfloor \sqrt{\tilde{M} + 2^{\ell-1}} \right\rfloor_{-\alpha i_0}. \quad (19.37)$$

The overall tabulation error δ_{i_0} can be rewritten

$$\delta_{i_0} = S_{i_0} - \sqrt{\tilde{M} + 2^{\ell-1}} + \sqrt{\tilde{M} + 2^{\ell-1}} - \sqrt{M} \quad (19.38)$$

The first term is the error of rounding to the nearest and is bounded by $2^{-\alpha i_0 - 1}$. The second term can be rewritten

$$\sqrt{\tilde{M} + 2^{\ell-1}} - \sqrt{M} = \frac{\tilde{M} + 2^{\ell-1} - M}{\sqrt{\tilde{M} + 2^{\ell-1}} + \sqrt{M}}. \quad (19.39)$$

As $|\tilde{M} + 2^{\ell-1} - M| \leq 2^{\ell-1}$ and both $\sqrt{\tilde{M} + 2^{\ell-1}} \geq 1$ and $\sqrt{M} \geq 1$, a bound of δ_{i_0} is

$$|\delta_{i_0}| \leq 2^{-\alpha i_0 - 1} + 2^{\ell-2} \quad (19.40)$$

and the constraint (19.29), rewritten $|\delta_{i_0}| \leq \rho 2^{-\alpha i_0}$ and combined with (19.40), yields the constraint on ℓ :

$$\ell \leq 2 + \log_2(\rho - 1/2) - \alpha i_0. \quad (19.41)$$

For $\beta = 4$ and $\gamma = 2$, hence $\rho = 2/3$, this means a valid value of ℓ is $\ell = -\alpha i_0 - 1$.

Note that with this tabulation, the value $S_{i_0} = 2$ is reached, just like with the previous initialization. This is slightly unfortunate as it means we need

$\alpha i_0 + 2$ bits to represent S_{i_0} . If we could ensure $1 \leq S_{i_0} < 2$ instead, we would save the leading bit of weight 2. However, this turns out not to be possible: using a rounding down instead of the round to nearest in (19.37) leads to a version of constraint (19.41) which is unfeasible as $\rho \leq 1$.

19.1.5.3 Integrating the Error due to Carry-Save Computation

Let t_W be the bit position at which we truncate the carry-save value of βW_i before converting it into standard binary. The possible carry that we ignore when performing this addition would have the weight 2^{t_W} : the approximate value $\beta \widetilde{W}_i$ that we will be using in the selection function is

$$\beta \widetilde{W}_i = \beta W_i + \delta \quad \text{with } \delta = 0 \text{ or } \delta = -2^{t_W}. \quad (19.42)$$

This is a one-sided approximation: $\beta \widetilde{W}_i \leq \beta W_i$. From (19.42), we also have $\beta W_i \leq \beta \widetilde{W}_i + 2^{t_W}$. Therefore, to ensure the constraints (19.33) and (19.34) which defined the digit selection boundaries, it is enough to ensure the new constraints:

$$2(d_{-i-1} - \rho)S_i + \beta^{-i-1}(d_{-i-1} - \rho)^2 \leq \beta \widetilde{W}_i \quad (19.43)$$

$$\beta \widetilde{W}_i + 2^{t_W} \leq 2(d_{-i-1} + \rho)S_i + \beta^{-i-1}(d_{-i-1} + \rho)^2. \quad (19.44)$$

In other words, the missing information from the ignored carry shrinks the selection interval. Figure 19.6 illustrates this, using a truncation of βW_i at position $t_W = -3$.

Although this figure was constructed programmatically using TikZ (thus, hopefully avoiding some possible human errors compared to a manual drawing), visual inspection cannot be trusted due to the tiny values in constants in (19.43) and (19.44).

Ercegovac and Lang [EL94; EL04] derive, from the previous equations, formulas for determining the parameters of digit selection. As the constant terms in the boundary lines $\beta^{-i-1}(d_{-i-1} + \rho)^2$ quickly decrease with iteration i (compare Figs. 19.4, 19.5, and 19.6), there exists a value i_0 such that the same selection function can be used for $i \geq i_0$. They find that for $i \geq i_0 = 3$ the following selection function parameters are valid:

- truncation of W_i to bit position $t_W = -4$, so that 4 bits of \widetilde{W}_i are input to the selection function,
- truncation of S_i to $t_S = -4$, inputting its 4 leading fractional bits to the selection function.

With $i_0 = 3$, the tabulation of S_3 out of the 8 leading bits of M provides an initialization of the iteration whose cost is comparable to the cost of subsequent iterations.

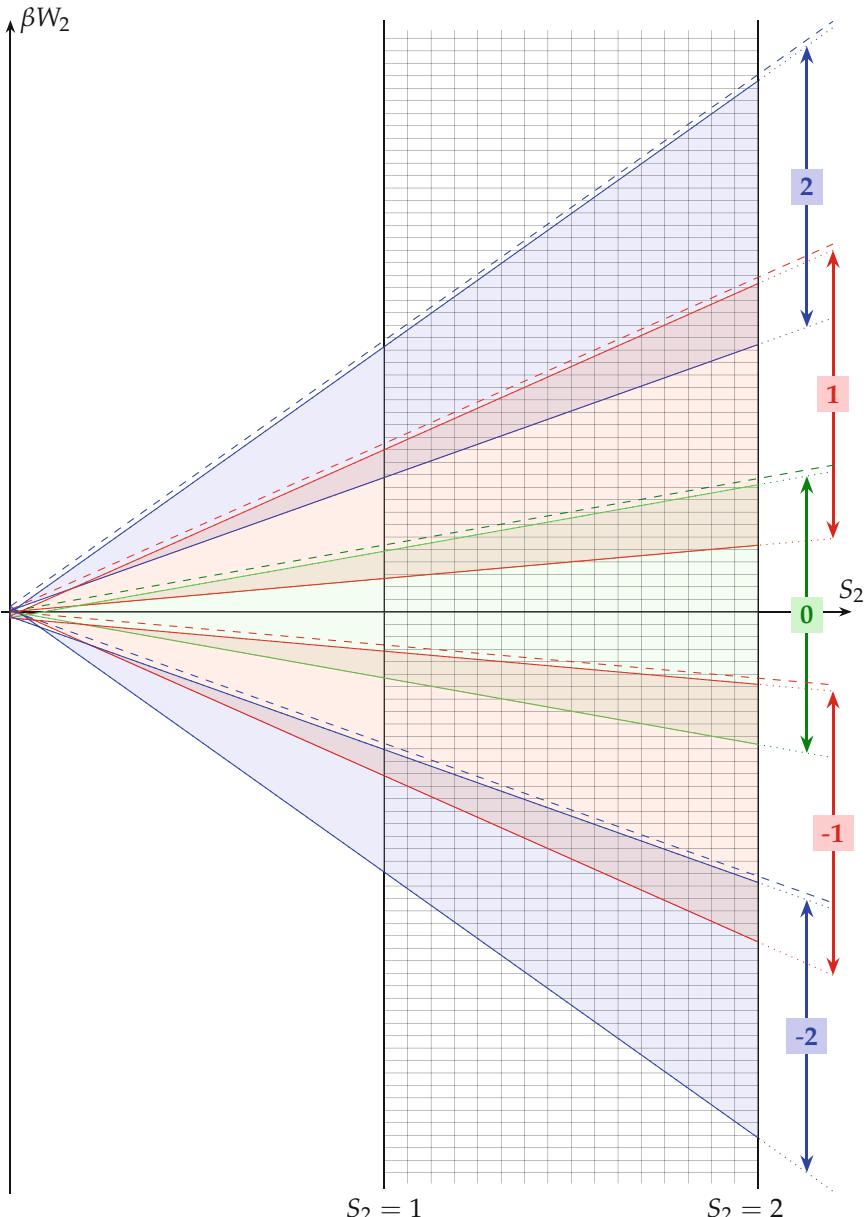


Fig. 19.6 P-D diagram at iteration $i = 2$ for the square root in radix 4 with $\gamma = 2$ and truncation $t = 3$. The dashed lines represent the boundaries without considering truncation.

A valid alternative to this analytical approach is a programmatic exploration of the parameter space for each i . Tentative points (t_W, t_S) of this space can be enumerated by increasing implementation cost $2^{t_W+t_S}$. When two solutions have the same $t_W + t_S$, the one with the smaller t_W will lead to the smaller latency. Then, for each (t_W, t_S) , an exhaustive enumeration checks if this point allows for a feasible selection function, using a careful implementation of (19.43) and (19.44) using safe arithmetic³ (e.g., using the proper directed rounding modes when rounding happens).

19.1.5.4 On-the-Fly Conversion of S_i

Tabulation-based initialization provides S_{i_0} in standard binary, but the following digits of S_i are provided in a redundant format. Conversion of S_i to standard binary is necessary before it can be truncated for input to the selection function. In $S_{i+1} = S_i + \beta^{-i-1}d_{-i-1}$, the addition is a concatenation if $d_{-i-1} \geq 0$ but an actual subtraction (with carry propagation possibly changing the leading bits) if $d_{-i-1} < 0$.

On FPGAs, it is possible to compute this subtraction using fast carry logic: it has a latency comparable to that of the computation of W_{i+1} using (19.27). However, it voids any area advantage of radix 4 over the non-restoring square root: we have twice as few iterations, but each iteration now requires two adders. To avoid this, it is possible to perform the complete conversion only once at the end and to use as input to the selection function a value \tilde{S}_i accurate to t_S only. This is similar to what we did for \tilde{W}_i . If t_S is even, we have $\tilde{S}_i = \sum_{j=0}^{-t_S/2} \beta^{-j} d_{-j}$. The worst-case sum of truncated digits is smaller in absolute value than $\sum_{j=t_S/2-1}^{\infty} 2\beta^{-j} = \rho\beta^{-t_S/2}$. The constraint to add to (19.43) and (19.43) is thus

$$\tilde{S}_i - \rho\beta^{-t_S/2} < S_i < \tilde{S}_i + \rho\beta^{-t_S/2} . \quad (19.45)$$

This possibly entails a more expensive selection function. A complete study of this trade-off remains to be done.

On ASICs, the latency of the addition computing S_i at each iteration is an issue. It is possible to avoid this latency by using the exact same on-the-fly conversion technique that was introduced for division (see Sect. 9.2.6, p. 280) and which we reproduce here. The idea is to maintain, along with each S_i , a pre-decremented version $S_i^\ominus = S_i - \beta^{-i}$, so that S_{i-1}^\ominus provides the result of the subtraction of β^{-i+1} when needed, i.e., when $d_{-i} < 0$.

The following recurrence implements this idea. It assumes that each digit d_j is written, using two's complement, as $d_j = -\beta d_j^\ominus + d_j^\oplus$ (where d_j^\oplus is a α -bit digit and d_j^\ominus is a single bit):

³ Note that the arithmetic engine used to construct our figures cannot be considered safe.

$$\text{if } d_{-i} > 0 \begin{cases} S_i = S_{i-1} + \beta^{-i} d_{-i}^{\oplus} \\ S_i^{\ominus} = S_{i-1} + \beta^{-i} (d_{-i}^{\oplus} - 1) \end{cases} \quad (19.46)$$

$$\text{if } d_{-i} = 0 \begin{cases} S_i = S_{i-1} \\ S_i^{\ominus} = S_{i-1}^{\ominus} + \beta^{-i} (\beta - 1) \end{cases} \quad (19.47)$$

$$\text{if } d_{-i} < 0 \begin{cases} S_i = S_{i-1}^{\ominus} + \beta^{-i} d_{-i}^{\oplus} \\ S_i^{\ominus} = S_{i-1}^{\ominus} + \beta^{-i} (d_{-i}^{\oplus} - 1) \end{cases} \quad (19.48)$$

The reader may check that all the additions in this recurrence are mere concatenations and that this recurrence maintains the invariant $S_i = \sum_j \beta^j d_j$ and $S_i^{\ominus} = S_i - \beta^{-i}$ (which can easily be shown by induction). This on-the-fly conversion requires in each iteration two multiplexers of the size of S_i and two registers in a pipelined design (for S_i and S_i^{\ominus}): it is fast but expensive in area.

19.2 Cube Root

The derivation of a recurrence computation of the cube root of a number [1, 8] is very similar to that of the square root [Tak01]. In the general case, the iteration works in radix β with a digit set that may be redundant (as introduced for division algorithms in Sect. 9.2.4, p. 272). Let us define the (increasingly accurate) partial cube root as

$$S_i = \sum_{j=0}^i \beta^{-j} d_{-j} \quad (19.49)$$

with $d_0 = 1$ and $S_0 = 1$. Let us also define the scaled remainder:

$$W_i = \beta^i (M - S_i^3). \quad (19.50)$$

Then remainder for iteration $i + 1$ can then be computed as

$$W_{i+1} = \beta^{i+1} (M - S_{i+1}^3) \quad (19.51)$$

$$= \beta^{i+1} \left(M - (S_i + \beta^{-i-1} d_{-i-1})^3 \right) \quad (19.52)$$

$$= \beta W_i - 3d_{-i-1} S_i^2 - 3\beta^{-i-1} d_{-i-1}^2 S_i - \beta^{-2i-2} d_{-i-1}^3. \quad (19.53)$$

The square S_i^2 that appears inside the recurrence can itself be incrementally computed by a parallel recurrence, so that (19.53) eventually resumes to only additions and digit-by-number multiplication.

Digit selection is performed (as for division and square root) by imposing on each W_i the constraint that is expected on the final remainder.

A complete study in the radix-2 case with the redundant digit set $\{-1, 0, 1\}$ was conducted by Pineiro et al. [Piñ+08]. In this case, digit selection can be decided considering only the 5 most significant digits of W_i . The implementation of (19.53) then requires four 4:2 compressor rows (see Fig. 7.21, p. 181), one 3:2 compressor row, and four multiplexers for the digit-by-number multiplications.

The same article [Piñ+08] also compares this approach with a multiplier-based one by the same team (all using ASIC models). In a nutshell, the digit recurrence is twice as slow but also twice as small, for comparable area-time delay. This result can probably be generalized as a rule of thumb.

As it explicitly computes the remainder, the digit recurrence also directly enables correct rounding, whereas a multiplier-based approach (e.g., using a generic polynomial approximator) only provides a faithful approximation. In the case of the cube root, deducing the correct rounding from this approximation requires cubing, which is likely to void the latency advantage over the digit recurrence.

19.3 2D and 3D Euclidean Norms

It is possible [TK00] to design an iterative digit-recurrence algorithm that directly evaluates $\sqrt{X^2 + Y^2 + Z^2}$ for $\frac{1}{2} \leq X < 1$, $0 \leq Y < 1$ and $0 \leq Z < 1$. Just like the division algorithms in Sect. 9.2.4, p. 272, the iteration works in radix β with a redundant digit set (as introduced in Sect. 2.4, p. 46). It computes in parallel two sequences:

- S_j is built to converge to $\sqrt{X^2 + Y^2 + Z^2}$ digit by digit, as

$$S_j = S_{j-1} + \beta^{-j} d_{-j} \quad (19.54)$$

where d_{-j} is a digit in the redundant digit set $\{-\gamma, \dots, \gamma\}$. As for SRT division, the main challenge is the proper selection of d_{-j} .

- W_j is a kind of partial remainder, defined as

$$W_j \approx \beta^{-j}(S_j - X^2 - Y^2 - Z^2). \quad (19.55)$$

If X , Y , and Z are fixed-point numbers, then W_j can be computed exactly. However, in [TK00], it is not an exact remainder but an accurate enough truncation – this truncation is particularly relevant if X , Y , and Z are floating-point numbers: in this case, the exact $X^2 + Y^2 + Z^2$ may require a very large amount of bits to represent when the exponent difference is large between X , Y , and Z .

In practice, W_{j+1} is computed incrementally out of W_j , this computation being obtained by substituting (19.54) in (19.55), as in the previous sections of this chapter.

One trick can be used here: the variables are first reordered such that $X \geq Y$ and $X \geq Z$. Thanks to this, the iteration can be initialized with $S_0 \approx X$, and the square of X never needs to be computed. The computations of Y^2 and Z^2 are performed incrementally (one line of partial products at each iteration) within the computation of W_{j+1} .

The derivation of the selection function determining d_{-j} is quite technical but follows the method used in textbooks [EL94; EL04] for division and square root. We refer the interested reader to the original article [TK00] for details.

In [TK00], the use of a redundant representation is mandatory, due to the fact that an approximate remainder is used. The simplest implementation is a radix-2 one with digit set $\{-1, 0, 1\}$ (thus, producing one bit of the result per iteration). The selection function inputs 6 bits of W_{j-1} to produce d_{-j} . The computation of W_{j+1} is a sum of five terms:

$$W_{j+1} = 2W_j + 2^{-3}(y_{-j-4}Y + z_{-j-4}Z) - 2d_{-j-1}S_j - 2^{-j-1}d_{-j-1}^2 \quad (19.56)$$

where the offset 2^{-3} compensated by the index $-j - 4$ is a technicality that captures the “accurate enough” approximation in (19.55). For the same reason, W_0 is initialized as $YY_{[3]} + ZZ_{[3]}$, where $Y_{[3]}$ denotes the 3-bit word formed by the 3 leading bits of Y . This sum can be computed in carry-save to avoid the carry propagation. Also, two versions of S_j are maintained to enable on-the-fly conversion of the result to binary (as introduced for division in Sect. 9.2.6, p. 280). Again, we refer the interested reader to the original article [TK00] for details.

For higher radices, the computation of W_{j+1} still adds five terms, but the digit-by-word multiplications are more expensive, as is the selection function. A radix-4 version with digit set $\{-2, -1, 0, 1, 2\}$ is detailed in the original article [TK00].

We are not aware of an implementation-based comparison of this digit-recurrence approach with a more naive approach summing the squares and then computing a square root. Obviously, the digit-recurrence approach will have a shorter latency. It also completely avoids the computation of X^2 . However, it computes the two other squares as plain products, missing the optimization opportunity explored in Chap. 14. The iterative computation of these products hidden in two additions of (19.56) is also less area-efficient than a global compressor computing a square. Therefore, the net benefit on area of this integrated digit recurrence is unclear.

To conclude this section, this algorithm can be extended to N -dimensional norms or simplified to a 2D Euclidean norm. An alternative way to compute a 2D norm using shifts and addition is the CORDIC algorithm in vectoring mode [EL04; Par10; Meh+09]. When unrolled in space, it requires roughly

$2n$ additions of n bits, along with $2n$ multiplexers. Here also, to the authors' knowledge, an implementation-based comparison remains to be done.

19.4 The E-Method for Evaluating Rational Functions

The E-method was introduced by Ercegovac [Erc77] as a generic digit-recurrence technique for evaluating a family of functions that can be expressed as the solution \mathbf{y} to a linear system $\mathbf{A}\mathbf{y} = \mathbf{b}$ for a non-singular matrix \mathbf{A} . The E-method computes an increasingly accurate approximation to $\mathbf{A}^{-1}\mathbf{b}$, one digit at a time. This digit recurrence can be viewed as a vector version of the digit-recurrence division algorithms of Sect. 9.2.

19.4.1 Digit Recurrence

The E-method incrementally computes an approximation $\mathbf{Y} = (Y_0 \ Y_1 \ \dots \ Y_n)^T$ to the exact solution \mathbf{y} using a radix- β representation: each component of this solution, Y_i for $i \in \{0, \dots, n\}$, is of the form

$$Y_i = \sum_{j=-1}^{-m} \beta^j d_{i,j} \quad (19.57)$$

where m is the desired precision of the result and the $d_{i,j} \in \{-\gamma, \dots, \gamma\}$ are digits in some redundant number system in radix β . Note that this assumes that a solution exists that can be written in this format (no nonzero digit of weight larger than β^{-1}). This constraint will be formalized in the sequel; it is similar to the constraint we have on the divisor for a digit-recurrence division to converge.

Let us note \mathbf{d}_j for $j \in \{-1, \dots, -m\}$ the vector formed by all the digits of weight β^j of \mathbf{Y} :

$$\mathbf{d}_j = (d_{0,j} \ d_{1,j} \ \dots \ d_{n,j})^T \quad (19.58)$$

The E-method is based on the following recurrence that computes the vector digits left to right, from $j = -1$ to $j = -m$:

$$\mathbf{W}_0 = \mathbf{b} \quad (19.59)$$

$$\mathbf{d}_j = \text{Sel}(\mathbf{W}_{j+1}) \quad \forall j \in \{-1, \dots, -m\} \quad (19.60)$$

$$\mathbf{W}_j = \beta(\mathbf{W}_{j+1} - \mathbf{A}\mathbf{d}_j) \quad \forall j \in \{-1, \dots, -m\} \quad (19.61)$$

Here, $\mathbf{W}_j = (W_{0,j} \ W_{1,j} \ \dots \ W_{n,j})^T$ is a vector of partial remainders, and Sel is a simple selection function that determines each element of the next digit vector \mathbf{d}_j by a simple rounding of the corresponding partial remainder element:

$$d_{i,j} = \begin{cases} \text{sign}(W_{i,j}) \times \left\lfloor |W_{i,j}| + \frac{1}{2} \right\rfloor & \text{if } W_{i,j} \leq \gamma \\ \text{sign}(W_{i,j}) \times \gamma & \text{otherwise} \end{cases} \quad (19.62)$$

The original publication [Erc77] defines more precisely the conditions (on \mathbf{A} , \mathbf{b} , and the number system) under which an iteration following Eqs. (19.59) to (19.62) converges to the exact solution \mathbf{y} of the linear system $\mathbf{Ay} = \mathbf{b}$. More precisely, it proves by induction that if there exists ζ and μ such that

$$\frac{1}{2} \leq \zeta < 1 \quad (19.63)$$

$$0 < \mu < \frac{1}{\beta} \left[1 - \frac{\zeta}{\gamma} (\beta - 1) \right] \quad (19.64)$$

$$||\mathbf{I} - \mathbf{A}|| \leq \mu \quad \text{using the norm } ||\mathbf{M}|| = \max_i \left(\sum_{k=0}^n |m_{ik}| \right) \quad (19.65)$$

$$||\mathbf{b}|| \leq \zeta \quad \text{using the norm } ||\mathbf{b}|| = \max_i |b_i| \quad (19.66)$$

then at each iteration j , we have

$$||\mathbf{W}_j - d_j|| \leq \zeta \quad (19.67)$$

and eventually

$$||\mathbf{Y} - \mathbf{y}|| < \beta^{-m} \quad . \quad (19.68)$$

In addition, if $\zeta > 1/2$, the redundancy of the number system can be exploited to relax (19.62): the digit selection can be decided based on an approximation $\widetilde{\mathbf{W}}_j$ defined as a truncation of \mathbf{W}_j to its most significand digits. We refer the reader to the original publication [Erc77] for the details.

19.4.2 Rational Approximation Compatible with the E-Method

A particularly interesting application, present in the original paper and later refined by Brisebarre et al. [Bri+08; Bri+18], is the evaluation of a numerical function $f(x)$ that can be approximated by a rational function:

$$R(x) = \frac{P_0 + P_1x + P_2x^2 + \cdots + P_nx^n}{1 + Q_1x + Q_2x^2 + \cdots + Q_nx^n} \quad \text{with } P_i, Q_i \in \mathbb{R} \quad (19.69)$$

The reader will check that if $\mathbf{y} = (y_0 \ y_1 \ \dots \ y_n)^T$ is the solution to the following linear system:

$$\begin{pmatrix} 1 & -x & & & (0) \\ Q_1 & 1 & -x & & \\ Q_2 & & 1 & -x & \\ \vdots & & & \ddots & \ddots \\ & & & & \ddots & \ddots \\ \vdots & (0) & & & 1 & -x \\ Q_n & & & & & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ \vdots \\ y_{n-1} \\ y_n \end{pmatrix} = \begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ \vdots \\ \vdots \\ P_{n-1} \\ P_n \end{pmatrix} \quad (19.70)$$

then $y_0 = R(x)$: the E-method, in this case, provides a recurrence that merges the two polynomial evaluations and the division of (19.69) in a single iteration.

The convergence conditions (19.65) and (19.66), in this case, are rewritten

$$\forall i \in \{1, \dots, n\} \quad |Q_i| + |x| < \mu \quad (19.71)$$

and

$$\forall i \in \{1, \dots, n\} \quad |P_i| < \zeta \quad . \quad (19.72)$$

As (19.71) involves x , this constraint defines the domain of convergence of the method with respect to the input x .

What do these constraints mean in practice?

In the special case of a polynomial function ($\forall i \ Q_i = 0$), there always exists some scaling by a power of two that will ensure that the constraints are satisfied. However, the E-method does not seem a particularly efficient way of evaluating polynomials. Besides, approximation by a rational function is more powerful than polynomial approximation since it may capture singularities in the function being approximated – this was discussed in Sect. 18.1.7.

For rational approximations in such cases, it can be shown [BM04] that the constraints (19.71) and (19.72) can be transformed, by scaling both the input and the output of the rational function by powers of two, into a constraint on the input interval. The relevance of the E-method then depends on the availability of an efficient range reduction. *Simple E-fractions* are defined [Bri+08] as rational fractions satisfying (19.71) and (19.72), possibly scaled by a power of two (on their output only). A method for finding simple E-fractions whose coefficients P_i and Q_i are machine-representable numbers is provided in [Bri+18]. This article also provides a reference implementation targeting FPGAs. The partial remainders are represented in standard two's complement binary, which is more efficient thanks to the fast carry hardware of FPGAs than the redundant representations used in the original works [Erc77] targeting ASICs.

The conclusion of this article [Bri+18] is that the E-method is particularly relevant for functions where a rational approximation is relevant and also that high radices (from $\beta = 8$ to $\beta = 32$) can lead to area-efficient FPGA implementations by exploiting the large architectural LUTs.

References

- [BM04] Nicolas Brisebarre and Jean-Michel Muller. “Functions Approximable by E-Fractions”. In: *Asilomar Conference on Signals, Circuits and Systems*. Vol. 2. IEEE, 2004, pp. 1341–1344 (cit. on p. 594).
- [Bri+08] Nicolas Brisebarre, Sylvain Chevillard, Miloš D. Ercegovac, Jean-Michel Muller, and Serge Torres. “An Efficient Method for Evaluating Polynomial and Rational Function Approximations”. In: *International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE, 2008, pp. 245–250 (cit. on pp. 593, 594).
- [Bri+18] Nicolas Brisebarre, George Constantinides, Miloš Ercegovac, Silviu-Ioan Filip, Matei Istoan, and Jean-Michel Muller. “A High Throughput Polynomial and Rational Function Approximations Evaluator”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2018, pp. 99–106 (cit. on pp. 593, 594).
- [EL04] Miloš D. Ercegovac and Tomás Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004 (cit. on pp. 585, 590).
- [EL94] Miloš D. Ercegovac and Tomás Lang. *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, Boston, 1994 (cit. on pp. 585, 590).
- [Erc77] Miloš D. Ercegovac. “A General Hardware-Oriented Method for Evaluation of Functions and Computations in a Digital Computer”. In: *IEEE Transactions on Computers C-26.7* (1977), pp. 667–680 (cit. on pp. 591, 592, 594).

- [Meh+09] Pramod K. Meher, Javier Valls, Tso-Bing Juang, K. Sridharan, and Koushik Maharatna. "50 Years of CORDIC: Algorithms, Architectures, and Applications". In: *IEEE Transactions on Circuits and Systems I : Regular papers* 56.9 (2009), pp. 1893–1907 (cit. on p. 590).
- [Par10] Behrooz Parhami. *Computer Arithmetic, Algorithms and Hardware Designs*. 2nd ed. Oxford University Press, 2010 (cit. on p. 590).
- [Piñ+08] José-Alejandro Piñeiro, Javier D. Bruguera, Fabrizio Lamberti, and Paolo Montuschi. "A Radix-2 Digit-by-Digit Architecture for Cube Root". In: *IEEE Transactions on Computers* 57.4 (2008), pp. 562–566 (cit. on p. 589).
- [Tak01] Naofumi Takagi. "A Digit-Recurrence Algorithm for Cube Rooting". In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 84.5 (2001), pp. 1309–1314 (cit. on p. 588).
- [TK00] Naofumi Takagi and Seiji Kuwahara. "A VLSI Algorithm for Computing the Euclidean Norm of a 3D Vector". In: *IEEE Transactions on Computers* 49.10 (2000), pp. 1074–1082 (cit. on pp. 589, 590).

Part IV

Example Composite Operators



20

CHAPTER 20

Fixed-Point Sine and Cosine

They believe the world is run by geometry, sire. All lines and angles and numbers. That sort of thing, sire—Dios frowned—can lead to some very unsound ideas.

Terry Pratchett, Pyramids

This chapter refines and compares three approaches for computing sines and cosines in hardware, with a focus of high-throughput pipelined architecture. The first approach is the classic CORDIC iteration, including a reduced iteration technique and fine optimizations in datapath width and latency. The second is an ad hoc architecture specifically designed around trigonometric identities. The third uses the generic table- and DSP-based polynomial approximation of Chap. 18.

The sine and cosine functions are pervasive in signal processing applications. Some applications only require the sine or the cosine of a value at some point (Fig. 20.1a and b), but a fused sine and cosine implementation (Fig. 20.1c) is interesting for two reasons. Firstly, many applications require both sine and cosine, for instance, to implement two-dimensional rotations. Secondly, many methods (actually derived from rotation considerations) compute both anyway.

20.1 Mathematical Background

Figure 20.2 illustrates the textbook definitions of the sine and cosine functions on the trigonometric circle. These functions (with radian argument) are plotted in Fig. 20.3.

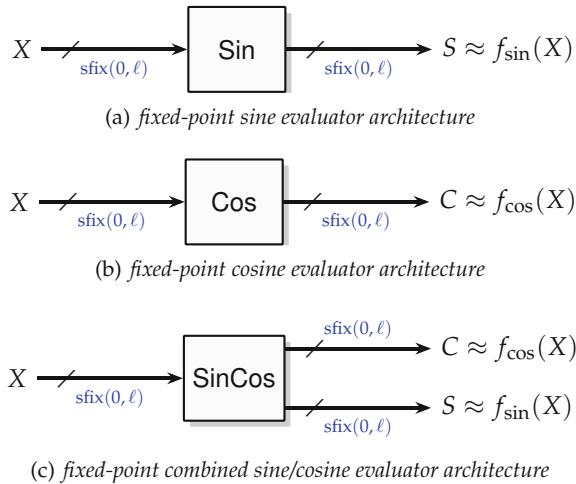


Fig. 20.1 Black box view of the functions studied in this chapter.

From these figures, a few classic trigonometric identities can be deduced to be used in this chapter (for more identities, see the books and websites listed in Chap. 16):

- By Pythagoras' theorem, we have

$$\cos^2 \theta + \sin^2 \theta = 1 \quad . \quad (20.1)$$

- By periodicity, we have

$$\sin(\theta + 2\pi) = \sin(\theta) \quad (20.2)$$

$$\cos(\theta + 2\pi) = \cos(\theta) \quad . \quad (20.3)$$

- By symmetry on the x axis, we have

$$\sin(-\theta) = -\sin(\theta) \quad (20.4)$$

$$\cos(-\theta) = \cos(\theta) \quad . \quad (20.5)$$

- By symmetry on the y axis, we have

$$\sin(\pi - \theta) = \sin(\theta) \quad (20.6)$$

$$\cos(\pi - \theta) = -\cos(\theta) \quad . \quad (20.7)$$

- By central symmetry, we have

$$\sin(\theta + \pi) = -\sin(\theta) \quad (20.8)$$

$$\cos(\theta + \pi) = -\cos(\theta) \quad . \quad (20.9)$$

- By symmetry on the main diagonal, we have

$$\sin\left(\frac{\pi}{2} - \theta\right) = \cos(\theta) \quad (20.10)$$

$$\cos\left(\frac{\pi}{2} - \theta\right) = \sin(\theta) \quad . \quad (20.11)$$

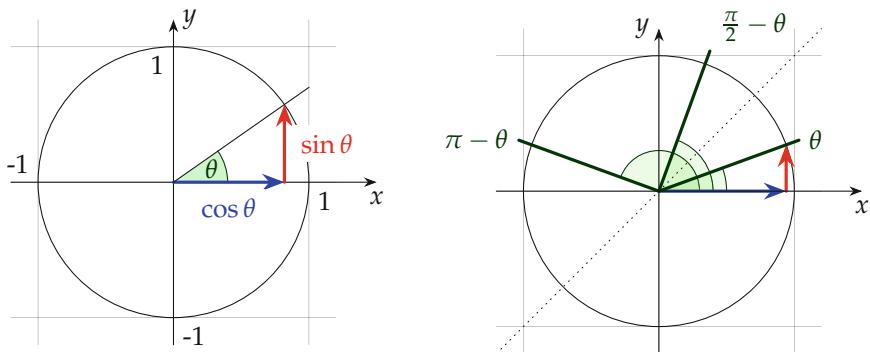


Fig. 20.2 The trigonometric circle and its symmetries.

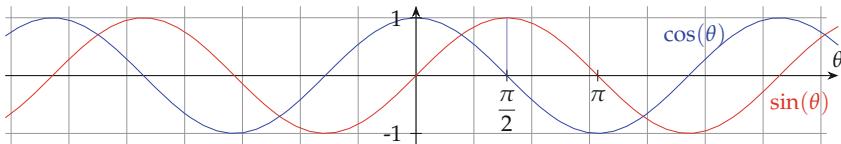


Fig. 20.3 Plots of the sine and cosine functions, with the argument in radian.

The sine and cosine functions can also be considered as the projections of the complex exponential on the real and imaginary axes:

$$e^{j\theta} = \cos(\theta) + j \sin(\theta) \quad . \quad (20.12)$$

Multiplying a complex number by $e^{j\theta}$ corresponds to a rotation of angle θ around the origin.

We also have the following identity on complex exponentials:

$$e^{j(a+b)} = e^{ja} \times e^{jb} \quad (20.13)$$

which simply expresses that the rotation of angle $a + b$ is obtained as the composition of the rotation of angle a and the rotation of angle b . Its real version is obtained by projection on the real and imaginary parts:

$$\begin{cases} \cos(a+b) = \cos(a)\cos(b) - \sin(a)\sin(b) \\ \sin(a+b) = \sin(a)\cos(b) + \cos(a)\sin(b) \end{cases} . \quad (20.14)$$

Finally, the Taylor expansions of sine and cosine are

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \quad (20.15)$$

$$= \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} \quad (20.16)$$

and

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots \quad (20.17)$$

$$= \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!} . \quad (20.18)$$

20.2 Fixed-Point Specification

20.2.1 Binary Angles

When dealing with machine numbers, radian arguments are not very convenient, in particular because the period 2π is an irrational number that will necessarily entail a systematic rounding error.

To avoid this, in this chapter, we choose instead to consider sine and cosine functions with *binary* angles: the functions considered are $\sin(\pi X)$ and $\cos(\pi X)$, where X is a signed (two's complement) number in $[-1, 1]$, represented in the $\text{sfix}(0, -\ell)$ format. As illustrated by Fig. 20.4, this specification maps the modulo- 2π trigonometric periodicity to the modulo-2 reduction behind two's complement representation. The input value -1 will be mapped to 1 by two's complement binary modulo arithmetic, which in terms of angles maps $-\pi$ to π : the interval $[-1, 1]$ indeed represents a full period of the function (Fig. 20.4).

Another point of view is that we can, using such an implementation, compute sines and cosines of numbers in a wider fixed-point range, for instance, having k integer bits, by simply dropping these bits, since they represent integer multiples of the period of the function. Some applications, in particular direct digital synthesis (DDS) [Cor04], are indeed based on the fact that these bits are dropped by the inherent modulo arithmetic of binary representations.

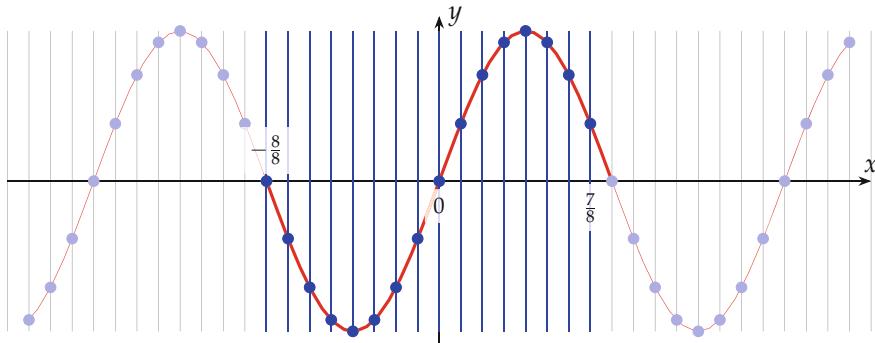


Fig. 20.4 Input discretization of $\sin(\pi X)$ with $\ell = -3$, with the fixed-point interval $[-1, 1]$ highlighted.

20.2.2 Scaled Output

On the output side, however, we have the problem already illustrated in Fig. 16.5 of Chap. 16. We need a value, 1, that does not belong to the open interval $[-1, 1]$ represented by the $\text{sfix}(0, -\ell)$ format. If we ignore the problem, we risk the catastrophic modulo effect of two's complement: the value 1 will be mapped to the value -1 (in Fig. 16.5, it was mapped to 0 since the output format was unsigned).

Among the solutions envisioned in Fig. 16.6 of Chap. 16, we choose to scale the output throughout this chapter (it is the solution illustrated by Fig. 16.6d). We decide that our operators should have a $\text{sfix}(0, -\ell)$ output format. Then, to be sure that a last-bit accurate architecture does not overflow this format, what the operators actually compute are the functions

$$f_{\sin}(x) = (1 - 2^{-\ell}) \sin(\pi x) \quad (20.19)$$

and

$$f_{\cos}(x) = (1 - 2^{-\ell}) \cos(\pi x) . \quad (20.20)$$

This is not a problem in practice, for several reasons. Firstly, in many signal processing applications, the shape is more important than the absolute amplitude. This is actually the reason why we prefer this scaling option to the “round toward zero” option. Secondly, applications can always reduce the corresponding systematic error by increasing the precision ℓ . Finally, as the sequel will show, for all proposed architectures, the scaling factor $(1 - 2^{-\ell})$ entails no additional cost. However, it should be clear that the last-bit accuracy specification will be with respect to these scaled functions f_{\sin} and f_{\cos} , not to $\sin(\pi x)$ and $\cos(\pi x)$.

20.2.3 Quick Overview of the Algorithms Compared in This Chapter

The well-known CORDIC algorithm, due to Volder [Vol59], computes both sine and cosine using a decomposition of the input angle into micro-rotations. These rotations are chosen in such a way that (20.14) can be computed only using additions and shifts. The details will be given in Sect. 20.4.

CORDIC was at the core of most early scientific calculators, and many CORDIC variations have been designed over time, initially mostly targeting ASIC, in two main directions:

1. to extend it to other functions (indeed, variations of CORDIC can compute exponential, logarithm, arctangent, and $\sqrt{x^2 + y^2}$),
2. to improve its speed, in particular using redundant arithmetic number systems to speed up each addition.

An extensive reference for all these variants is Muller's textbook [Mul16]. A review for the 50 years of CORDIC can be found in [Meh+09]. The contribution of the present book will only be a focus on the accuracy issue, including both approximation and rounding errors. Section 20.4 will present an instance of CORDIC computing just right for sine and cosine. The additions are performed using plain carry-propagate adders, which should be the preferred approach on FPGAs [VKP02], thanks to their fast carry logic. Indeed, CORDIC has been well studied on FPGAs [And98] and has been present in vendor core generators almost from the start.

CORDIC only uses the configurable logic of FPGAs, but modern FPGAs also include embedded memories and DSP blocks. These resources can also be used to evaluate sine and cosine, using Eq. (20.14) with a coarser decomposition of an input angle into a sum of two smaller angles. A good decomposition of θ (good because it is for free) is $\theta = \theta_h + \theta_l$ where θ_h is the number formed from the k leading bits of θ and θ_l is formed of the remaining lower bits, so that $\theta_l < 2^{-k}$. This enables the use of an acceptable table for $\sin(\theta_h)$ and $\cos(\theta_h)$ and a polynomial of small degree, typically Taylor, for $\sin(\theta_l)$ and $\cos(\theta_l)$. The tables exploit the embedded memories, while the polynomials and the implementation of (20.14) exploit embedded multipliers. This idea will be exploited in Sect. 20.5. Other decompositions have been suggested [Gal86; Mul16].

A variant of CORDIC, called “reduced iterations CORDIC” [THH89], uses a similar idea. It first rotates the input angle, using CORDIC micro-rotation, until the remaining angle θ_r is small enough that its sine and cosine can be approximated (with good enough accuracy) as $\sin \theta_r \approx \theta_r$ and $\cos \theta_r \approx 1$. This ensures that a final rotation of angle θ_r can be computed, using (20.14), with just two small multiplications and two additions.

The third technique, studied in Sect. 20.6, is simply the application to the sine or cosine functions of the generic polynomial approximation techniques of Chap. 18.

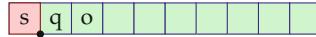


Fig. 20.5 The input number X in $\text{sfixed}(0, \ell)$ format.

Before exposing these techniques, Sect. 20.3 overviews the common argument reductions that benefit to the three techniques presented in this chapter.

20.3 Argument Reduction

The first argument reduction is based on symmetries on the unit circle. The three leading bits of X are called, respectively (Fig. 20.5),

- $X_0 = s$ for *sign*,
- $X_{-1} = q$ for *quadrant*,
- $X_{-2} = o$ for *octant*.

Let us call the remaining bits $Y \in \left[0, \frac{1}{4}\right)$, and let us define

$$Y' = \begin{cases} Y & \text{if } o = 0 \\ \frac{1}{4} - Y & \text{if } o = 1 \end{cases} \quad (20.21)$$

such that $Y' \in \left[0, \frac{1}{4}\right)$.

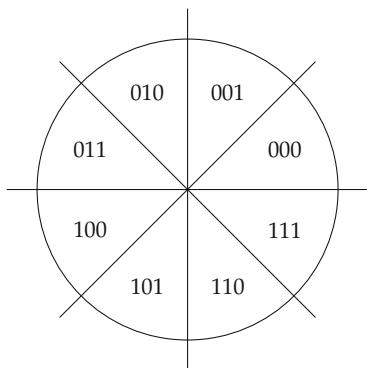
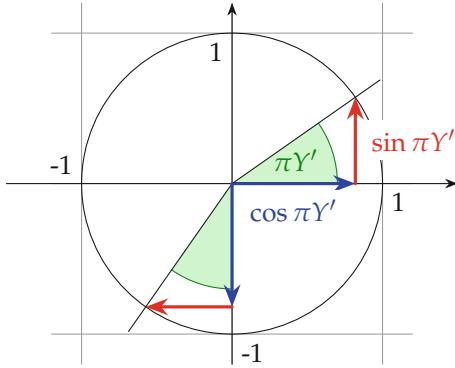
Figure 20.6 shows how the trigonometric identities listed in Sect. 20.1 are used to reduce the problem to computing $\sin(\pi Y')$ and $\cos(\pi Y')$ for $Y' \in \left[0, \frac{1}{4}\right)$.

20.4 CORDIC Computing Just Right

20.4.1 Description of the Algorithm

The core operation in the classic CORDIC algorithm is the rotation of a vector. A generic rotation of a vector by an angle θ can be described by the operation

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \underbrace{\begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}}_{=R} \begin{pmatrix} x \\ y \end{pmatrix} \quad (20.22)$$

(a) Decomposition of the unit circle by the (s, q, o) bits

(b) Illustration of the range reduction in the 101 quadrant

<i>sqo</i> Reconstruction	<i>sqo</i> Reconstruction
$000 \begin{cases} \sin(\pi X) = \sin(\pi Y') \\ \cos(\pi X) = \cos(\pi Y') \end{cases}$	$100 \begin{cases} \sin(\pi X) = -\sin(\pi Y') \\ \cos(\pi X) = -\cos(\pi Y') \end{cases}$
$001 \begin{cases} \sin(\pi X) = \cos(\pi Y') \\ \cos(\pi X) = \sin(\pi Y') \end{cases}$	$101 \begin{cases} \sin(\pi X) = -\cos(\pi Y') \\ \cos(\pi X) = -\sin(\pi Y') \end{cases}$
$010 \begin{cases} \sin(\pi X) = \cos(\pi Y') \\ \cos(\pi X) = -\sin(\pi Y') \end{cases}$	$110 \begin{cases} \sin(\pi X) = -\cos(\pi Y') \\ \cos(\pi X) = \sin(\pi Y') \end{cases}$
$011 \begin{cases} \sin(\pi X) = \sin(\pi Y') \\ \cos(\pi X) = -\cos(\pi Y') \end{cases}$	$111 \begin{cases} \sin(\pi X) = -\sin(\pi Y') \\ \cos(\pi X) = \cos(\pi Y') \end{cases}$

Fig. 20.6 Trigonometric argument reduction for binary angles.

The matrix R is called rotation matrix. The term $\cos(\theta)$ can first be factored out:

$$R = \cos(\theta) \begin{pmatrix} 1 & -\tan(\theta) \\ \tan(\theta) & 1 \end{pmatrix}. \quad (20.23)$$

We can now split a rotation by θ into several rotations by smaller angles α_i , called micro-rotations, such that

$$\theta = \sum_{i=0}^{\infty} \alpha_i. \quad (20.24)$$

If we now choose $\alpha_i = \pm \arctan 2^{-i}$, the rotation matrix of each micro-rotations becomes

$$R' = \cos(\alpha_i) \begin{pmatrix} 1 & \mp 2^{-i} \\ \pm 2^{-i} & 1 \end{pmatrix}. \quad (20.25)$$

Table 20.1 CORDIC angles of micro-rotations.

i	2^{-i}	$\alpha_i = \arctan 2^{-i}$ (in degree)
0	1	45
1	0.5	26.57
2	0.25	14.04
3	0.125	7.13
4	0.0625	3.58
5	0.03125	1.79
6	0.01563	0.9
7	0.00781	0.45
8	0.00391	0.22

The product by the matrix $\begin{pmatrix} 1 & -2^{-i} \\ 2^{-i} & 1 \end{pmatrix}$ is very cheap to implement in hardware, since it involves only additions and shifts.

However, without the constant $\cos(\alpha_i) = \frac{1}{\sqrt{1+2^{-2i}}}$, it is not a rotation: it also scales by this factor the module of the point being rotated (Fig. 20.7). In the CORDIC algorithm, this is not a problem: the product of all these scaling factors is precomputed, and the original vector is scaled by its inverse.

As the initial vector for computing the sine and cosine is $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ it can be performed by just changing the initialization.

The limitation to angles of $\alpha_i = \pm \arctan 2^{-i}$ may seem limiting, but, in fact, due to the sublinear behavior of the arctan function, any angle θ between 0° and 99.88° (1.7433 rad) can be represented using (20.24). The first elements of α_i are given in Table 20.1. The reader may convince himself that this limit is quickly approximated by summing the angles in Table 20.1.

With the range reduction discussed in Sect. 20.3, we only need a maximum angle of 45° ($\pi/4$); therefore, we can start the iteration at $i = 1$. There is no hardware benefit, though: the implementation of this range reduction is essentially the step $i = 0$ of CORDIC.

Now, the sine and cosine can be obtained using CORDIC by computing in parallel three values in each iteration i (starting from $i = 1$, assuming the range reduction introduced above):

- An angle Z_i , which is initialized to Y' and brought to 0 by a sequence of micro-rotations of angle $\pm \arctan(2^{-i})$.
- A coordinate vector (C_i, S_i) , starting with a point on the x axis and rotated in the opposite direction by the same angle $\arctan(2^{-i})$. As Z_i converges to 0, C_i and S_i converge to the cosine and sine of Y .

The complete CORDIC iteration is thus

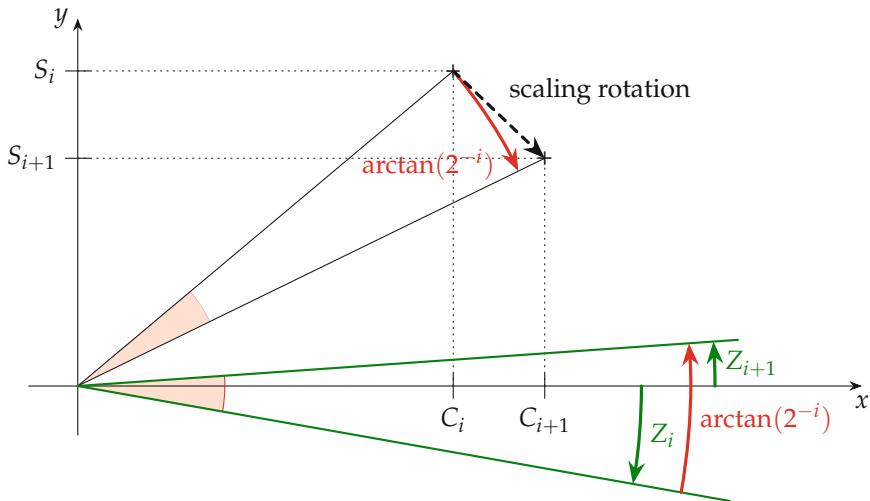


Fig. 20.7 Illustration of one CORDIC micro-rotation step (here, $i = 2$ hence $\arctan(2^{-i}) \approx 14$ degrees). Since $Z_i < 0$, we have $d_i = -1$.

$$\begin{cases} C_1 = \frac{1}{\prod_{i=1}^n \sqrt{1 + 2^{-2i}}} \\ S_1 = 0 \\ Z_1 = Y' \quad (\text{the reduced argument}) \end{cases} \quad (20.26)$$

$$\forall i \geq 1 \begin{cases} d_i = +1 \text{ if } Z_i > 0, \text{ otherwise } -1 \\ C_{i+1} = C_i - 2^{-i} d_i S_i \\ S_{i+1} = S_i + 2^{-i} d_i C_i \\ Z_{i+1} = Z_i - d_i \arctan(2^{-i}) \end{cases} \quad (20.27)$$

Each step of (20.27) can be implemented by the architecture shown in Fig. 20.8.

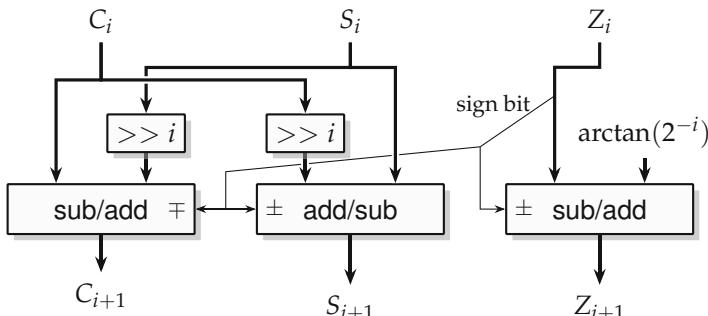


Fig. 20.8 Abstract architecture for one CORDIC iteration.

Note that textbooks often mention an iteration starting at $i = 0$, which ensures convergence for the full angle range. The constant used in this case, C_0 , is different from the constant C_1 in the equation above.

We consider in the following an unrolled (fully pipelined) CORDIC operator. In this case, the values of $\arctan(2^{-i})$ are constant inputs to the additions. Equation (20.27) works for radian arguments, but adapting it to our specification is simply a matter of scaling each constant by $\arctan(2^{-i})$, so it entails no cost. Similarly, the output scaling $(1 - 2^{-\ell})$ is merged at no cost in the initial value C_1 in (20.26).

In this equation, n is the total number of iterations and is yet unknown: the iteration should stop as soon as the result is accurate enough. To address this question, we need to define properly the error analysis of the CORDIC architecture which is addressed next.

20.4.2 Overall Error Budget

Let us call

- C_i and S_i the values computed by Eqs. (20.27) and (20.26) in infinite precision,
- \tilde{C}_i and \tilde{S}_i the values of C_i and S_i actually computed by the architecture,
- C and S the final values output by the architecture.

Following the methodology introduced in Chap. 3, we need to evaluate \tilde{C}_i and \tilde{S}_i using an internal precision that is g bits larger than the output precision. In the following, we only show the derivation for the sine datapath, since the cosine is identical.

For any input Y' , overall error δ_{total} will be the sum of the approximation error, the datapath rounding errors, and the final rounding error:

$$\delta_{\text{total}} = S - \sin Y' \quad (20.28)$$

$$= \underbrace{(S - \tilde{S}_n)}_{= \delta_{\text{final round}}} + \underbrace{(\tilde{S}_n - S_n)}_{= \delta_{\text{round}}} + \underbrace{(S_n - \sin Y')}_{= \delta_{\text{approx}}} \quad (20.29)$$

As usual, we can bound the final rounding error as one half-ulp of the result:

$$\delta_{\text{final round}} \leq 2^{\ell-1} \quad (20.30)$$

and our purpose is now to design an architecture in such a way that $\delta_{\text{round}} + \delta_{\text{approx}} < 2^{\ell-1}$, which will ensure last-bit accuracy (δ_{total} strictly smaller than one ulp of the result).

20.4.3 Approximation Error and Number of Iterations

The approximation error δ_{approx} is defined as the error that we would have if (20.26) and (20.27) were implemented with infinite accuracy. Back of the envelope, for large i Taylor gives $\arctan(2^{-i}) \approx 2^{-i}$, hence iteration i adds to Z_i a value that is (in absolute value) close to 2^{-i} , while it adds to C_i and S_i a value that is smaller than 2^{-i} , since C_i and S_i are themselves smaller than 1. In other words, the accuracy of C_i and S_i is improved at least by 1 bit at each iteration.

To achieve last-bit accuracy, let us therefore fix $n = -\ell + 2$. This ensures that $\delta_{\text{approx}} \leq 2^{\ell-2}$, leaving an error budget of $2^{\ell-2}$ (one quarter of an ulp of the result) for the accumulation of all the rounding errors.

20.4.4 Rounding Errors

As usual, the rounding errors will be controlled by adding g guard bits to the internal computation datapath: let us now assume that all the computations are performed on fixed-point formats with least significant bit (LSB) $2^{\ell-g}$. Let us also define

$$u = 2^{\ell-g} \quad (20.31)$$

the value of the unit in the last place (ulp) on this datapath.

On the Z_i datapath, each iteration adds a constant that is the correct rounding of the actual $\arctan(2^{-i})$ to the precision u . This adds an absolute error bounded by $u/2$ to the error of the previous iteration; therefore, the accumulated error after i iterations is bounded by $i \cdot u/2$.

For the C_i and S_i datapaths, we first have to discuss the choice introduced in Sect. 3.1.4 between truncation and correct rounding. Figure 20.9a shows the alignment of the two terms to be added.

The first option, truncation, is illustrated by Fig. 20.9b. It computes

$$\tilde{S}_{i+1} = \tilde{S}_i + \left\lfloor 2^{-i} d_i \tilde{C}_i \right\rfloor_{\ell-g}. \quad (20.32)$$

The rounding error in this case is

$$\delta_i^{(\text{add})} = \tilde{S}_{i+1} - (\tilde{S}_i + 2^{-i} d_i \tilde{C}_i) \quad \text{with} \quad |\delta_i^{(\text{add})}| < 2^{\ell-g} \quad (20.33)$$

Another option, illustrated by Fig. 20.9, is to round the sum to the nearest by adding one half-ulp:

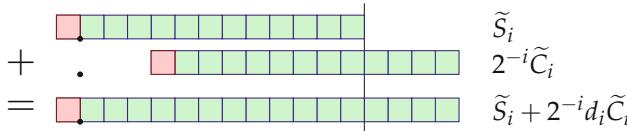
$$\tilde{S}_{i+1} = \left[\tilde{S}_i + \left\lfloor 2^{-i} d_i \tilde{C}_i \right\rfloor_{\ell-g-1} \right]_{\ell-g} \quad (20.34)$$

This actually involves *two* rounding errors bounded by $u/2$ each:

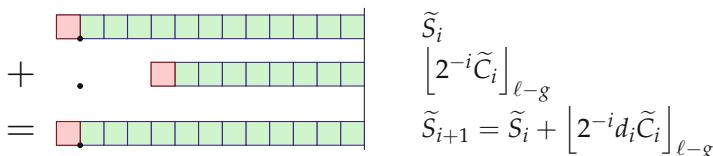
- the truncation of $2^{-i}\tilde{C}_i$ to precision $2^{\ell-g-1}$,
- the rounding to the nearest of the sum from precision $2^{\ell-g-1}$ to precision $2^{\ell-g}$.

Therefore, this option incurs no net improvement on accuracy, while the hardware cost is slightly higher (each adder larger by 1 bit).

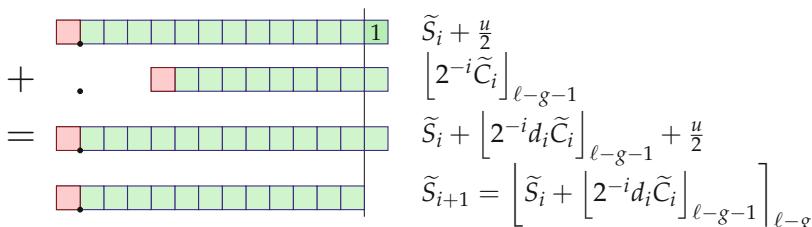
In both cases, the error contribution of one step of S_i or C_i computation is $\overline{\delta_i^{(\text{add})}} = 1u$.



(a) Alignment and exact addition



(b) CORDIC step with truncation



(c) CORDIC step with the addition rounded to the nearest

Fig. 20.9 One addition on the S and C datapath of the CORDIC algorithm.

This is but the rounding error of one step. Let us now see how these errors accumulate. We may rewrite

$$\delta_{\text{round},i+1} = \tilde{S}_{i+1} - S_{i+1} \quad (20.35)$$

$$= \tilde{S}_i + \left\lfloor 2^{-i} d_i \tilde{C}_i \right\rfloor_{\ell-g} - (S_i + 2^{-i} d_i C_i) \quad (20.36)$$

$$= \tilde{S}_i + \left\lfloor 2^{-i} d_i \tilde{C}_i \right\rfloor_{\ell-g} - (\tilde{S}_i + 2^{-i} d_i \tilde{C}_i) \\ + (\tilde{S}_i + 2^{-i} d_i \tilde{C}_i) - (S_i + 2^{-i} d_i C_i) \quad (20.37)$$

$$= \left\lfloor 2^{-i} d_i \tilde{C}_i \right\rfloor_{\ell-g} - 2^{-i} d_i \tilde{C}_i \\ + (\tilde{S}_i - S_i) + 2^{-i} d_i (\tilde{C}_i - C_i) . \quad (20.38)$$

Assuming by symmetry the same error bound $\bar{\delta}_{\text{round},i} = 1u$ for both $\tilde{S}_i - S_i$ and $\tilde{C}_i - C_i$, the cumulated error bound (expressed in ulps) verifies the following recurrence:

$$\bar{\delta}_{\text{round},i+1} = 1 + (2^{-i} + 1)\bar{\delta}_{\text{round},i} . \quad (20.39)$$

This computation only holds if the ideal sequence (C_i, S_i) and the approximate one $(\tilde{C}_i, \tilde{S}_i)$ indeed “take the same decisions,” i.e., share the same values of d_i . However, it may happen that $Z_i > 0$ but $\tilde{Z}_i = 0$, due to rounding errors applied to a value of Z_i very close to 0. In this case, the ideal sequence would choose $d_i = 1$, while the actual one will chose $d_i = 0$, and the computations will diverge from here. The proper way to manage such issues in an error analysis is *not* to attempt to compute accurately enough so that the approximate sequence takes the same decisions as the ideal one. This could require a very high accuracy with prohibitively high cost. Instead, the proper point of view is to consider, for the purpose of error analysis, the ideal sequence that takes the same decisions as the approximate one. Indeed, in the previous corner case where the two sequences would diverge at step i , both choices of d_i will eventually lead to a convergence to the same correct limit values.

How do we initialize this recurrence? We have $\tilde{S}_1 = S_1 = 0$. However, the initial value C_1 from (20.26) must be rounded to the working precision $2^{\ell-g}$. The corresponding error $\bar{\delta}_{\text{round},1} = \tilde{C}_1 - C_1$ can be computed with high accuracy using multiple-precision software or upper-bounded by $u/2$ (the current choice in the FloPoCo implementation).

Then, the recurrence (20.39) is evaluated by software up to

$$\bar{\delta}_{\text{round}} = \bar{\delta}_{\text{round},n} . \quad (20.40)$$

All that remains is to rewrite the constraint that $\bar{\delta}_{\text{round}}$ ulps should be smaller than our rounding error budget:

$$\bar{\delta}_{\text{round}} \times 2^{\ell-g} < 2^{\ell-2} . \quad (20.41)$$

Hence, the smallest value of g that ensures last-bit accuracy is

$$g = 2 + \lceil \log_2(\bar{\delta}_{\text{round}}) \rceil . \quad (20.42)$$

One final remark, we have so far assumed that the range reduction as well as the final reconstruction introduced no approximation nor rounding error. Indeed, the computations presented in Fig. 20.6 can all be implemented exactly in fixed point. However, as soon as we have guard bits, it is slightly faster to implement both the possible subtraction $Y' = \frac{1}{4} - Y$ and the final negation by bitwise complement, using a row of XORs: this saves one carry propagation. The price to pay is one single ulp of error for each negation, which usually does not even change the value of g , and hence incurs no architectural overhead.

20.4.5 Reduced Z_i Datapath

A further optimization, possible in the case of an unrolled implementation, is to compute just right the iteration on Z_i . Indeed, each iteration roughly removes one most significant bit from the angle Z_i . In other words, the MSB of Z_i is $-i$. It is therefore possible to reduce the size of the adders on the Z_i datapath (the rightmost adder/subtractor in Fig. 20.8) by 1 bit at each iteration. This observation almost divides by 2 the cost of the Z_i datapath, thus reducing the total area by about 1/6th. On FPGAs, synthesis tools are able to pack the C_i and S_i lines of Eq. (20.27) as one LUT per bit while still using the fast carry logic. The LUT consumption is thus very predictable, close to $2.5(-\ell + g)^2$.

In principle, this datapath optimization on Z_i could also slightly reduces the critical path: in the early iterations, the critical data dependency of one iteration to the next one is $Z_i \rightarrow d_i \rightarrow Z_{i+1}$. If d_i is known earlier because Z_i is shorter, it becomes possible to launch the computation of C_{i+1} and S_{i+1} as soon as the needed bits of C_i and S_i have been computed, i.e., before the end of iteration i . Then the carry propagations of the two successive iterations can progress in parallel. In the later iterations, as C_i and S_i are being shifted further, the gain is smaller, which is unfortunate because it is when d_i can be computed the fastest.

20.4.6 Replacing Half of the CORDIC Iterations with a Small Multiplier

In this variant, the CORDIC iteration is stopped as soon as the remaining rotation can be computed, with sufficient accuracy, by

$$\begin{cases} C_{\text{final}} = C_i + \pi Z_i Y_i \\ S_{\text{final}} = S_i - \pi Z_i X_i \end{cases} \quad (20.43)$$

This is really an application of (20.14) with the Taylor approximations $\cos(Z_i) \approx 1$ and $\sin(Z_i) \approx \pi Z_i$ – remember that Z_i is a binary angle. It therefore introduces a new approximation error term in the error analysis, due to the corresponding Taylor remainders. A rule of thumb is that this error term is at most close to $\frac{Z_i^2}{2}$, thus smaller than $2^{\ell-3}$ as soon as $Z_i < 2^{\ell/2-1}$, or for $i = \lfloor -\ell/2 \rfloor + 1$: this is where we may safely stop the CORDIC iteration. We leave as an exercise to the reader to add a safe and proper bound of the approximation error to the error analysis and then to add the rounding error contributions of the two multiplications. A solution to this exercise is in the FloPoCo source code.

Then the size of Z_i is $\lfloor -\ell/2 \rfloor + 1 + g$ bits. It may be multiplied by π using a FixRealCM constant multiplier (see Sect. 12.3) with an absolute error bound of one ulp. Then this product must be multiplied with both C_i and S_i , using two multipliers.

We may as well truncate C_i and S_i before the products to the same size as πZ_i : this saves almost half the hardware at the cost of a contribution of one ulp to the overall error. Thus, (20.43) can be implemented with only two multipliers, each of size slightly larger than $-\ell/2$ bits.

In practice, for instance, the signed 18-bit multiplications that are the common denominator of most FPGAs can be used for values of ℓ up to -24 .

The computation of the initial scaling factor must be modified accordingly, as well as the error analysis. These and other technical details can be found in the FloPoCo code.

20.5 An Architecture Based on Tables and Multipliers

The architecture presented here was developed to exploit the block memories and embedded multipliers of recent FPGAs. Interestingly, on FPGAs,¹ even when synthesized only using logic, its LUT cost is comparable to CORDIC while its timing is much better [DIS13]. The reason for this is that it expresses more bit-level parallelism and that both tables and multipliers can still be implemented efficiently in LUTs.

¹ A proper comparison on VLSI targets with CORDIC variants speeded up by the use of redundant number systems (see [Meh+09] and references therein) remains to be done.

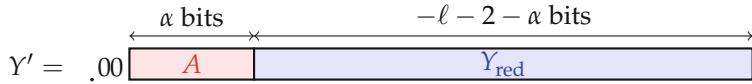


Fig. 20.10 Second argument reduction.

20.5.1 Algorithm

Here, we further split our octant angle Y' into its α most significant bits A and its lower bits $Y_{\text{red}} \in [0, 2^{-2-\alpha})$, as illustrated by Fig. 20.10. We keep α as an open parameter for now, and its value will be determined in the sequel.

A is used to address a table storing $\sin(\pi A)$ and $\cos(\pi A)$. Meanwhile, the sine and cosine of πY_{red} are evaluated by first computing $Z = \pi Y_{\text{red}}$ and then by using the Taylor series (20.15) and (20.15) of $\sin(Z)$ and $\cos(Z)$, respectively. The addition of two angles can then be applied by the rotation formulas (20.14) which allow us to recover the values of $\sin(\pi Y')$ and $\cos(\pi Y')$:

$$\sin(\pi Y') = \sin(\pi A) \cos(Z) + \cos(\pi A) \sin(Z) \quad (20.44)$$

$$\cos(\pi Y') = \cos(\pi A) \cos(Z) - \sin(\pi A) \sin(Z) \quad (20.45)$$

Here, a Taylor series can be preferred over the generic polynomial approximation of Chap. 18 for two reasons. The first is that the sine series is odd and the cosine series is even, so their evaluation requires half as many multiplications for a given degree. The second is that up to terms of degree 4, the coefficients are powers of two (whose computation is for free) or powers of two multiplied by $1/3$ (which is also very cheap to compute, as seen in Chap. 13). For instance, if we limit the two Taylor series for $\sin(Z)$ and $\cos(Z)$ to their two leading terms, we get

$$\sin(Z) \approx Z - \frac{Z^3}{6} \quad (20.46)$$

$$\cos(Z) \approx 1 - \frac{Z^2}{2} \quad . \quad (20.47)$$

Then, since $\cos(Z)$ is always very close to 1, it is interesting to rewrite the product $\sin(\pi A) \times \cos(Z)$ as $\sin(\pi A) - \sin(\pi A) \times \frac{Z^3}{6}$. This replaces a large multiplier with a significantly smaller one and an addition. The same holds, of course, for $\cos(\pi A) \times \cos(Z)$. Therefore, instead of (20.44) and (20.45), we prefer to use the following formulas:

$$\sin(\pi Y') \approx \sin(\pi A) - \sin(\pi A) \times \frac{Z^2}{2} + \cos(\pi A) \times (Z - \frac{Z^3}{6}) \quad (20.48)$$

$$\cos(\pi Y') \approx \cos(\pi A) - \cos(\pi A) \times \frac{Z^2}{2} - \sin(\pi A) \times (Z - \frac{Z^3}{6}) \quad (20.49)$$

Figure 20.11 shows the resulting architecture.

20.5.1.1 Architectural Constraints from Approximation Error Analysis

A first constraint on α is that the approximation in (20.46) and (20.47) must be accurate enough. The corresponding approximation errors are

$$\delta_{\text{Taylor}(\sin)} = \sin Z - (Z - \frac{Z^3}{6}) \quad (20.50)$$

$$\delta_{\text{Taylor}(\cos)} = \cos Z - (1 - \frac{Z^2}{2}) \quad . \quad (20.51)$$

Here, $\delta_{\text{Taylor}(\cos)}$ is roughly $\frac{Z^4}{24}$, the first neglected term of the Taylor series. For $Z = \pi Y_{\text{red}} < 3.15 \cdot 2^{-2-\alpha} < 2^{-\alpha}$ (its MSB will be $m_Z = -\alpha$), this Taylor remainder can actually be bounded as $\delta_{\text{Taylor}(\cos)} < 2^{-4\alpha}/24 < 2^{-4\alpha-4}$ (the margin considered in all these upper bounds is more than enough to absorb all the higher-order terms). Similarly, $\delta_{\text{Taylor}(\sin)} < 2^{-5\alpha}/120 < 2^{-5\alpha-6}$.

Back to (20.44) and (20.45), we see that the terms $\sin(Z)$ and $\cos(Z)$ are multiplied by a sine or a cosine, each bounded by 1. The approximation errors $\delta_{\text{Taylor}(\sin)}$ and $\delta_{\text{Taylor}(\cos)}$ will thus not be amplified. Therefore, as soon as these errors are strictly smaller than one half-ulp of the result, it is possible to build a last-bit accurate architecture. This translates to the following constraint: $2^{-4\alpha-4} < 2^\ell$, or

$$\alpha > -\frac{\ell}{4} - 1 \quad . \quad (20.52)$$

20.5.2 Rounding Error Analysis and Implementation Details

It remains to decide the sizing of each wire and of each operator on the architecture depicted in Fig. 20.11. This includes the various truncations that have to be applied to obtain an architecture that computes just right. As usually (see Chap. 3), let us assume an intermediate format (before the final rounding) with a precision $2^{\ell-g}$ where g is a number of guard bits. There will be many error terms in this architecture, and one principle of computing just right is to balance them so that no part of the architecture computes ridiculously accurately compared to the rest. In practice, we ensure that each

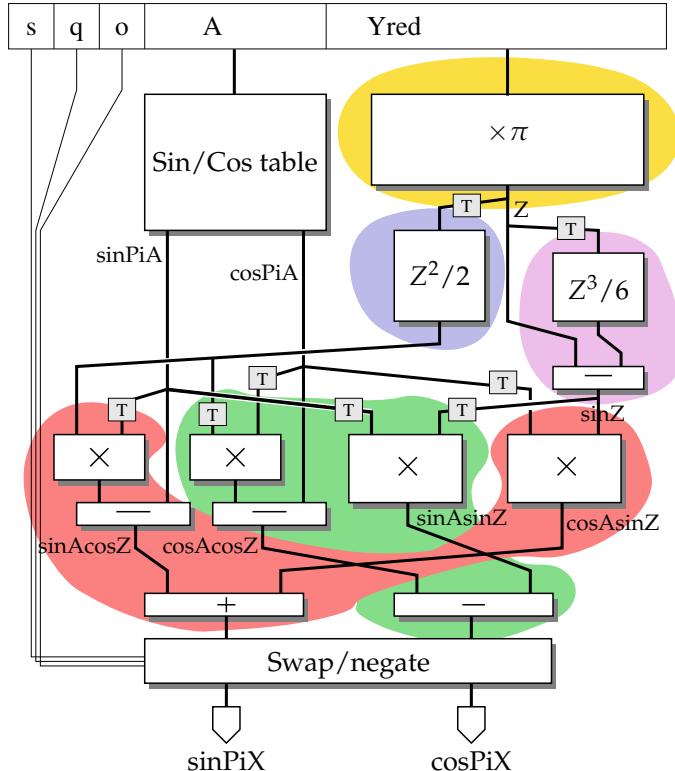


Fig. 20.11 A DSP- and table-based architecture. Each colored bubble is a bit heap.

error term being summed to the final error is of the order of $2^{\ell-g}$. This entails a constraint on the LSB of each term.

Let us consider first the multipliers, taking as an example the multiplier computing $\sin(\pi A) \times \frac{Z^2}{2}$. It computes the product of two approximate terms. Following the generic error analysis for multipliers of Sect. 8.1.3, both inputs should be truncated to the same size. In details, we instantiate (8.10) with $X = \sin(\pi A)$ (hence, $m_X = 0$ and ℓ_X to be determined) and $Y = \frac{Z^2}{2}$ (hence, $m_Y = -2\alpha - 1$ since $m_Z = -\alpha$ and ℓ_Y to be determined). Then, the constraint (8.10) becomes

$$\ell - g \approx -2\alpha - 1 + \ell_X \approx 0 + \ell_Y, \quad (20.53)$$

and solving this constraint defines the truncations to apply: before this multiplication, $\sin(\pi A)$ should be truncated to $\ell_X = \ell - g + 2\alpha + 1$, and $\frac{Z^2}{2}$ should be truncated to $\ell_Y = \ell - g$. Of course, the MSB of the product will be $-2\alpha - 1$.

When the same $\sin(\pi A)$ is multiplied with $Z - \frac{Z^3}{6}$, it will be truncated to a different size. All these truncations are illustrated by the small boxes labeled \boxed{T} in Fig. 20.11. Finally, when it is added to the final sine bit heap, it needs to be accurate to its precision, which is $\ell - g$.

Tables 20.2 and 20.3 attempt to synthesize all the formats and truncations used in the architecture. Truncations are also applied before each power term (the square is a product, and we leave the analysis for the cube as an exercise to the reader). Each term must be computed to its highest precision in these tables and then truncated when used in other places.

Table 20.2 Formats of the internal data on the architecture of Fig. 20.11. All these fixed-point formats, except X and R , are unsigned. Further truncations are given in Table 20.3.

Variable	MSB	LSB
X, R	1	ℓ
$\sin(\pi A), \cos(\pi A)$	0	$\ell - g$
Y_{red}	$-\alpha - 2$	ℓ or $\ell - g$
$Z, Z - \frac{Z^3}{6}$	$-\alpha$	$\ell - g$
$\frac{Z^2}{2}$	$-2\alpha - 1$	$\ell - g$
$\frac{Z^3}{6}$	$-3\alpha - 2$	$\ell - g$

Table 20.3 Truncations in the architecture of Fig. 20.11.

Product	m_X	ℓ_X	m_Y	ℓ_Y	m_P	ℓ_P
$\sin(\pi A) \times \frac{Z^2}{2}$	0	$\ell - g + 2\alpha + 1$	$-2\alpha - 1$	$\ell - g$	$-2\alpha - 1$	$\ell - g$
$\sin(\pi A) \times (Z - \frac{Z^3}{6})$	0	$\ell - g + \alpha$	$-\alpha$	$\ell - g$	$-\alpha$	$\ell - g$

20.5.3 Architectural Details

$Z^2/2$ is computed using either a truncated squarer or simply a tabulation. The latter is more accurate (one half-ulp error bound) than the former (one full ulp).

$Z^3/6$ is very small: its size in bits is $-3\alpha - 2 - \ell + g$ with $\alpha > -\frac{\ell}{4} - 1$ by (20.52). Roughly speaking, its size is one fourth the input size, plus the guard bits. This value can be simply tabulated for most practical values of the precision ℓ .

On FPGAs, the multipliers can be implemented in logic only for small sizes, or as truncated DSPs for larger sizes, or as a combination of both.

In addition, we have another constraint on FPGAs: the look-up tables and block RAM should be used fully when used. For some sizes, this will allow for a larger value of α “for free” than the smallest one defined by (20.52). Having a larger α will reduce the size of the multipliers and may even entail that an order-1 architecture (neglecting the $Z^2/2$ term altogether) is accurate enough. The FloPoCo implementation attempts to exploit this, and the reader is referred to its source code for more details.

For larger sizes, we designed an ad hoc architecture that computes $Z - Z^3/6$ inside a single bit heap. Only a few bits need to be divided by 3, using a variation of the divider by 3 of Chap. 13. An example of bit heap thus obtained is shown in Fig. 20.12.

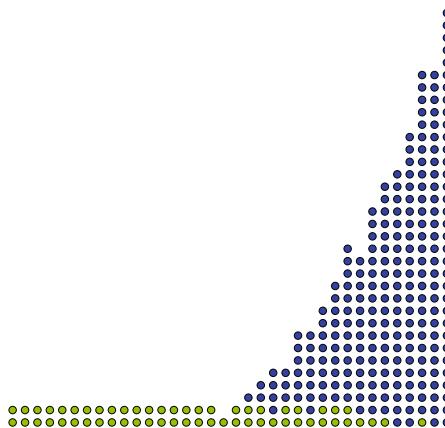


Fig. 20.12 The bit heap computing $z - z^3/6$ for a 40-bit sine/cosine operator.

20.5.4 Computing the Number of Guard Bits

The detailed error analysis is not interesting enough to be detailed here in full. Rounding errors due to one operation can be composed carefully, and overall, this fixed-point architecture loses up to 15 ulps to rounding/truncation errors: one half-ulp for each table, one ulp for each truncation \boxed{T} and for each truncated multiplier or squarer, one ulp for the initial computation of $1/4 - y$ by XORing, and one ulp that captures all the higher-order terms and all the error terms that have been scaled down. Therefore, the value of g that enables last-bit accuracy is 4 – remark that here it does not depend on ℓ . This error analysis is assuming the worst-case architectural variant (e.g., $\frac{Z^2}{2}$

and g . $\frac{Z^3}{6}$ computed, not tabulated, etc.). In the more favorable cases, a few ulps are replaced with half-ulps, but this is not enough to win one guard bit.

20.5.5 Special Cases for Small Sizes

The current FloPoCo implementation also includes several special cases that simply resort to tabulation for very small sizes. By order of increasing input size, it will

- use plain tables on the whole domain,
- use plain tables on an octant, with the first argument reduction,
- use the second argument reduction but with first-order approximations $\sin(Z) \approx Z$ and $\cos(Z) \approx 1$,
- use a second-order approximation with $\sin(Z) \approx Z$ and $\cos(Z) \approx 1 - Z^2/2$,
- use the third-order approximations $\sin(Z) \approx Z - Z^3/6$ and $\cos(Z) \approx 1 - Z^2/2$ as shown in Fig. 20.11.

Of course, the value of g is adapted in consequence.

20.6 Architecture Using a Generic Polynomial Evaluator

The last option explored in this chapter is a wrapper of the generic polynomial evaluator of Chap. 18, combined with the range reduction of Sect. 20.3.

In addition to the argument X , this operator inputs a bit $\sin/\overline{\cos}$ that determines if the sine or the cosine should be computed. Here, X is only reduced to one quadrant: $Y' \in [0, 1/2)$. The polynomial evaluator computes $\sin(\pi Y')$ for $Y' \in [0, 1/2)$. The proper output is reconstructed out of $\sin(\pi Y')$ and the bits s, q and $\sin/\overline{\cos}$.

20.7 Comparison and Discussion

Hands on: Sine and Cosine Architectures

The two methods computing both sine and cosine can be tested with the following command:

```
flopoco FixSinCos
```

while the polynomial wrapper can be tested by

```
flopoco FixSinOrCos
```

We leave it to the reader to select which architecture is best for his context. A detailed comparison of unrolled CORDIC versus the table and multiplier approach of Fig. 20.11 was conducted in the context of FPGAs [DIS13]. In this experiment, all the tables and all the multipliers were implemented in FPGA logic (not using DSP blocks and RAM blocks) to allow for a simple comparison. With this setup, unrolled CORDIC and the table and multiplier approach consume very comparable resources. However, CORDIC is much slower (from a factor 2 for 16 bits to a factor 3 for 32 bits in logic delay). One may conclude that an unrolled CORDIC makes little sense in this context. Faster variants of CORDIC using redundant number systems may catch up with delay but will consume more resources. This is the main reason why they were not studied in detail in this chapter.

However, when the application can accommodate a lower throughput, CORDIC is still relevant in its iterative variant, which requires several (un-pipelined) cycles to complete but consumes much lower resources. Note, however, that it requires a bit more logic than what is shown in Fig. 20.8: there, both the shifts and the $\arctan(2^{-i})$ constants are hardwired and cost no logic. In an iterative version, two barrel shifters are needed to implement the shifts, and a table must hold the constants $\arctan(2^{-i})$. This consumes more logic than the three adders of Fig. 20.8. Besides, the Z datapath reduction of Sect. 20.4.5 is no longer possible. In this sense, the unrolled CORDIC is more efficient in terms of throughput per area than the iterative one.

It is unclear how these results will translate to ASIC. On the one hand, the fast carry logic on the FPGA side favors CORDIC, all the more as the CORDIC iteration hardware can be densely packed in modern FPGA structures. On the other hand, tables and small multipliers are also very efficiently implemented with FPGA logic.

References

- [And98] Ray Andraka. “A survey of CORDIC algorithms for FPGA based computers”. In: *Sixth international symposium on Field Programmable Gate Arrays*. ACM. 1998, pp. 191–200. (cit. on p. 604).
- [Cor04] Lionel Cordesses. “Direct Digital Synthesis: A Tool for Periodic Wave Generation (Part 1)”. In: *IEEE Signal Processing Magazine* 21.4 (2004), pp. 50–54. (cit. on p. 602).
- [DIS13] Florent de Dinechin, Matei Istoan, and Guillaume Sergent. “Fixed-Point Trigonometric Functions on FPGAs”. In: *SIGARCH Computer Architecture News* 41.5 (2013), pp. 83–88. (cit. on pp. 614, 621).

- [Gal86] Schmuel Gal. "Computing elementary functions: A new approach for achieving high accuracy and good performance". In: *Accurate Scientific Computations, LNCS 235*. Springer Verlag, 1986, pp. 1–16. (cit. on p. [604](#)).
- [Meh+09] Pramod K. Meher, Javier Valls, Tso-Bing Juang, K. Sridharan, and Koushik Maharatna. "50 Years of CORDIC: Algorithms, Architectures, and Applications". In: *IEEE Transactions on Circuits and Systems I : Regular papers* 56.9 (2009), pp. 1893–1907. (cit. on pp. [604, 614](#)).
- [Mul16] Jean-Michel Muller. *Elementary Functions, Algorithms and Implementation*. 3rd ed. Birkhäuser Boston, 2016. (cit. on p. [604](#)).
- [THH89] D. Timmermann, H. Hahn, and B. Hostika. "Modified CORDIC Algorithm with Reduced Iterations". In: *Electronics Letters* 25.15 (1989), pp. 950–951. (cit. on p. [604](#)).
- [VKP02] Javier Valls, Martin Kuhlmann, and Keshab K. Parhi. "Evaluation of CORDIC Algorithms for FPGA Design". In: *Journal of VLSI Signal Processing* 32.3 (2002), pp. 207–222. (cit. on p. [604](#)).
- [Vol59] Jack Volder. "The CORDIC Computing Technique". In: *IRE Transactions on Electronic Computers* EC-8.3 (1959), pp. 330–334. (cit. on p. [604](#)).



CHAPTER 21

Floating-Point Accumulation and Sum of Products

Fortunately, the stupendous progress of mechanic art in modern times makes it comparatively easy. Thanks to the Piano Electro-Reckoner, the most complex calculations can be made in a few seconds.

Jules Verne, *In The Year 2889*

Summing many independent terms or products is a very common operation. Scalar product, matrix-vector and matrix-matrix products, tensor products, and linear digital filters are based upon sums of products. Other applications, such as numerical integration or Monte Carlo simulations, also involve sums of many independent terms. For all these applications, this chapter describes a family of floating-point accumulators that accept one floating-point summand at each cycle and compute the sum with arbitrary accuracy.

This chapter addresses the computation of a sum $\sum_{i=1}^N X_i$ or a sum of products $\sum_{i=1}^N X_i Y_i$ where X_i and Y_i are floating-point numbers.

If N is small and constant, one may build a tree of floating-point adders. When N is very large or when it is not known at the construction of the circuit, the solution is an iterative accumulator, as illustrated by Fig. 21.1. On this figure, a register holds the current (running) partial sum. A reset signal starts a new summation.

A first idea is to use, for the \oplus box of Fig. 21.1, a standard floating-point adder. A problem is that floating-point adders have long latencies, from three to five cycles in ASIC, up to tens of cycles in an FPGA. This is explained by the complexity of their architecture (see Fig. 11.5, p. 337, or Fig. 11.6, p. 341). This long latency l means that an accumulator based

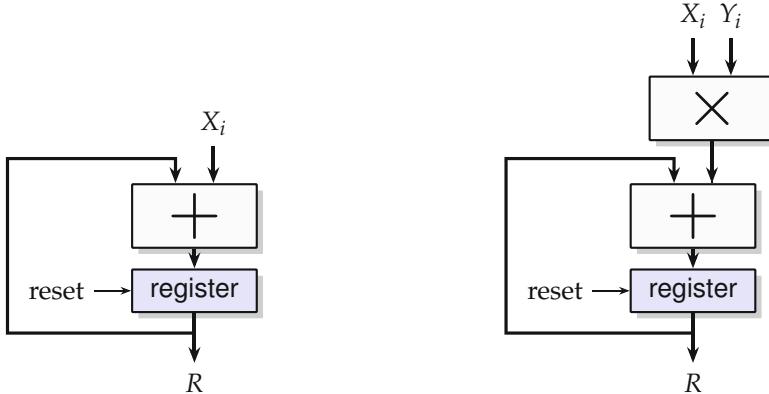


Fig. 21.1 Functional view of iterative accumulation (left) and sum of products (right).

on an FP adder will either add one number every l cycles. Alternatively, its pipeline will compute l independent sub-sums, which then have to be added together somehow, adding to the complexity and cost of the accumulation (unless at least l accumulations can be interleaved, as in large matrix operations [ZP05; Bod+06]).

This long latency is essentially due to the significand shifts in an accumulator built out of a floating-point adder (again see Fig. 11.5). These shifts are in the critical path of the loop of Fig. 21.1. Massive hardware speculation [Lut19] can reduce the latency (see Sect. 11.1.4, p. 340) but at a very high hardware cost.

Another issue with accumulation based on a floating-point adder is that it can be very inaccurate [Mul+18]. Even if the error due to the computation of one summand X_i can be bounded, the error due to the summation will grow with N . A simple solution is to use, for the adder of Fig. 21.1, a floating-point format with a larger significand than the summands (the conversion of a floating-point input to a larger format is cheap and always errorless). However, this solution further increases the latency on the loop's critical path.

This chapter addresses the construction of a floating-point accumulator that is not based on a standard floating-point adder. A key idea, illustrated by Fig. 21.2, is to keep the accumulator register in fixed-point, so that it does not need to be shifted. The shifts only concern the summands and are kept out of the loop's critical path.

Section 21.1 discusses the specification of such a fixed-point accumulator, in such a way that overflows are avoided and that the accumulation process is arbitrarily accurate and even exact. Combining such an accumulator with a modified, errorless FP multiplier, the same properties are achieved for an iterative sum-of-product operator. Section 21.2 then provides simple

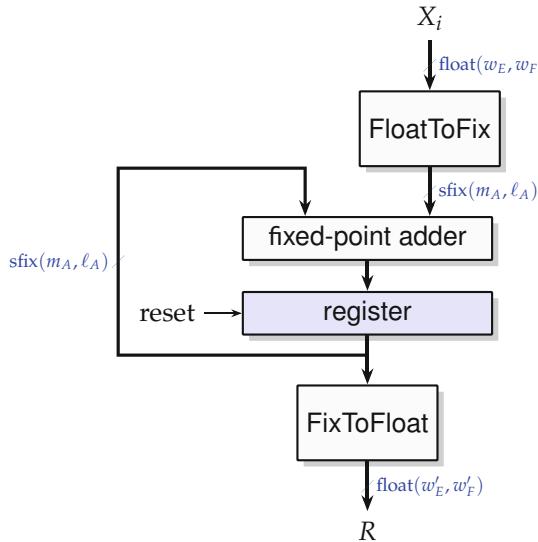


Fig. 21.2 Keeping the accumulator in fixed point.

techniques to build such an accumulator, in such a way that it may input a floating-point datum every cycle, for an arbitrary frequency.

21.1 Motivation and Parametrization

21.1.1 Exact Floating-Point Sum

Figure 21.3 shows the binary representation of a sum $\sum_{i=1}^N X_i$ when performed exactly (without any rounding nor overflow). This figure only shows the significands (here, all positive and all normal), each shifted according to its exponent and summed.

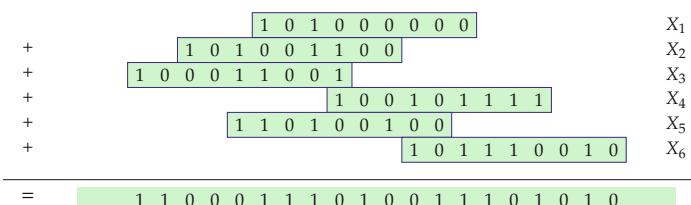


Fig. 21.3 Exact sum of a sequence of floating-point numbers.

Table 21.1 Parameters of an exact accumulator for IEEE 754 inputs.

Format of X_i	Extreme MSB and LSB of X_i	Min. acc. size
binary16	$(w_E, w_F) = (5, 10)$	$(e_{\max}, e_{\min} - w_F) = (15, -24)$
binary32	$(8, 23)$	$(127, -149)$
binary64	$(11, 52)$	$(1023, -1074)$

The exponents of a floating-point format are bounded between e_{\min} and e_{\max} (defined as a function of exponent size w_E and fraction size w_F for IEEEfloat and Nfloat by Table 2.4, p. 55). Therefore, in the sum $\sum_{i=1}^N X_i$, each summand can be converted, without rounding nor overflow, into a fixed-point format whose most significant bit (MSB) is e_{\max} and whose least significant bit (LSB) is $e_{\min} - w_F$. Numerical values for the standard IEEE 754 formats are given in Table 21.1.

Note that this table only depends on the parameters (w_E, w_F) of the input format. Figure 21.2 allows for a different output format with parameters (w'_E, w'_F) : this output format is only used in the construction of the **FixToFloat** box of Fig. 21.2.

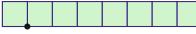
21.1.2 Exact Floating-Point Sum of Products

We recall from Chap. 11 that the exact product of two positive floating-point numbers $X = 2^{E_X} \cdot M_X$ and $Y = 2^{E_Y} \cdot M_Y$ is

$$XY = 2^{E_X + E_Y} \cdot M_X \times M_Y . \quad (21.1)$$

A figure very similar to Fig. 21.3 would therefore represent the exact sum of products $\sum_{i=1}^N X_i Y_i$. In this case, we do not shift significands but exact products $M_X \times M_Y$ of two significands (see Fig. 21.4), shifted by the sum of the exponents. Remark that there is no need to normalize a product before shifting it to place: $E_X + E_Y$ is the exponent associated with the unnormalized product $M_X \times M_Y$ and can be used to control the shift.

In a sum of products $\sum_{i=1}^N X_i Y_i$, the product of two floating-point numbers can always be converted without error into a fixed-point format whose MSB is $2e_{\max} + 1$ (the $+1$ due to the case when the significand product $M_X \times M_Y$ is larger than 2; see Fig. 21.4) and whose LSB is $2e_{\min} - 2w_F$. Numerical values for the standard IEEE 754 formats are given in Table 21.2.

A significand: 

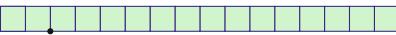
The exact product
of two significands: 

Fig. 21.4 Fixed-point view of floating-point significands and their unnormalized product for $w_F = 7$.

Table 21.2 Parameters of an exact accumulator of exact products of IEEE 754 inputs.

	Format of X_i and Y_i extreme MSB and LSB of $X_i Y_i$	min. acc. size
	$(w_E, w_F) \quad (2e_{\max} + 1, 2e_{\min} - 2w_F)$	
binary16	(5, 10) (31, -48)	80
binary32	(8, 23) (255, -298)	554
binary64	(11, 52) (2047, -2148)	4196

21.1.3 Kulisch's Universal Exact Accumulator

From this observation, Capello and Miranker [CM88] and then Kulisch and his team [KK88; Ker+94; Kul13] suggested that floating-point units could be enhanced with a fixed-point accumulator that covers this range. This would ensure that sums and sums of products of floating-point data can be performed without any rounding error. The last column in Table 21.2 gives the minimum width of such an accumulator.

The LSB of this large accumulator is clearly fixed: no bit will ever be created at a position lower than this LSB. Note that subnormals are included, since all the subnormals have the e_{\min} exponent.

The MSB, however, is a more open question. Indeed, even though no floating-point product may ever have bits in position larger than $2e_{\max} + 1$, their sum may. This overflow situation can be detected and reported as an infinity. Besides, in the common use case where the sum is to be finally converted to a floating-point number, any sum with bits higher than position e_{\max} will be converted to an infinity. However, it may happen that some intermediate sum overflows while the final result would be representable, due to positive and negative summands cancelling each other. To absorb such *spurious overflows*, Kulisch suggested to add even more bits at the MSB of the floating-point format. To quantify how many bits are needed, consider the worst-case situation of a 5 GHz computer accumulating one product per cycle for 10 years. Spurious overflow can be avoided altogether in this case by adding $\lceil \log_2(365 \times 24 \times 60 \times 60 \times 5 \cdot 10^9) \rceil = 58$ bits at the MSB of the fixed-point accumulator. This almost doubles the size of a binary16 exact accumulator but is a minor enlargement for an exact binary64 accumulator whose minimum size is already 4196 bits (see Table 21.2).

So far, we have reasoned only about positive summands: the management of signed inputs is an implementation issue that will be discussed in Sect. 21.2.

A Fallacy: Application-Level Accuracy Using Exact Accumulators

The reader should be aware that the exact or accurate accumulators discussed here do not guarantee high application-level accuracy: the result of an exact accumulation is no more accurate than the data that was thrown into it.

For instance, in Fig. 21.3, if X_3 was the result of some rounding operation (as most floating-point numbers are), this rounding discarded some bits that were to the right of its LSB (here, at bit positions -4 and lower). Therefore, in terms of application-specific accuracy, the “exact sum” probably holds no meaningful bits below position -4 .

What the exact accumulator ensures is only that the summation process itself is exact, and in a perfectly scalable way, since it does not depend on N . This property is already an important element of a complete error analysis, as introduced in Chap. 3. Moreover, exactness also brings useful properties such as the associativity of the sum (classic floating-point addition is not associative). With an exact accumulator, the sum will be the same whatever the order of arrival of the X_i .

21.1.4 Application-Specific Parameterization

In an application-specific context, it is often possible to tailor an exact accumulator to an application. The goal here is to achieve satisfying application-level accuracy while using smaller values of the accumulator size than those given in Table 21.2.

A first obvious application-specific optimization is the case when the summands are not products but plain floating-point numbers. In the (co)processor hardware context envisioned in Kulisch’s works, simple sums must use the same exact accumulator hardware as sums of products (for instance, by setting all the Y_i to 1.0 in this case). In an application-specific context, however, an accumulator designed for simple sums can be twice as small as one designed for sums of products (compare Tables 21.1 and 21.2). This situation is also very easy to detect by an HLS compiler [Ugu+20].

A second, even more powerful way of designing application-specific accumulators is to tailor the MSB and the LSB of the accumulator itself, as illustrated by Fig. 21.5:

- m_A is the position of the most significant bit (MSB) of the accumulator. If the maximal expected sum is smaller than 2^{m_A} , no overflow ever occurs.

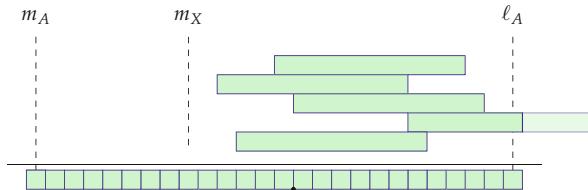


Fig. 21.5 Parameters of a fixed-point accumulator (here, $m_A = 13$, $\ell_A = -12$, $m_X = 5$).

- ℓ_A is the position of the least significant bit of the accumulator. It controls the final accuracy of the summation process.
- $w_A = m_A - \ell_A + 1$ is the width of the accumulator.
- m_X is the maximum expected position of the MSB of a summand. m_X may be equal to m_A , but very often one is able to tell that each summand is much smaller in magnitude than the final sum. In such cases, providing $m_X < m_A$ will save hardware by reducing the size of the input shifter.

For many applications, values of m_X , m_A , and ℓ_A can be determined a priori, using a rough error analysis or software profiling.

A representative example is the computation of the inductance of a set of coils by numerical integration [Cre+07]. In this application, physics tells us that the sum will be less than 10^5 (using arbitrary units due to factoring out some physics constants). Software profiling showed that the absolute value of a summand was always between 10^{-2} and 2.

Converting to bit positions and adding two orders of magnitude (or 7 bits) for safety in all directions, this defines $m_A = \lceil \log_2(10^2 \times 10^5) \rceil = 24$, $m_X = 8$, and $\ell_A = -w_F - 15$. For $w_F = 23$ (single precision), we conclude that an accumulator stretching from $\ell_A = -23 - 15 = -38$ to $m_A = 24$ will be able to absorb all the additions with virtually no rounding error.

None of the summands encountered in the profiling simulation will add bits lower than 2^{-38} . As profiling is by definition not exhaustive, it may happen that some summands in an actual run violate this bound, and the hardware should handle this case (contrary to an exact accumulator, where by design no summand can have bits smaller than $2^{e_{\min} - w_F}$). However, it will be rare, and the rounding error in this case (2^{-38}) will be tiny compared to the final result.

The accumulator is also large enough to ensure it never overflows, and we have left two orders of magnitude for spurious overflow, which is (by physics standard) a wide margin.

In this example, the accumulator size is $w_A = 24 + 38 + 1 = 63$ bits, much smaller than the $127 + 149 + 1 = 177$ bits of an exact accumulator.

Remark that in this example, only ℓ_A depends on w_F . The MSB m_A is strictly related to the physics of the problem, regardless of the precision used to simulate it. The parameter m_X depends on the physics but also on the

discretization step chosen for the numerical integration. The parameter ℓ_A allows a designer to manage the accuracy/area trade-off for an accumulator.

In the previous application example, the accumulation size N is not taken into account. Another possible approach to setting ℓ_A is the following: if the result is required with an accuracy 2^ℓ and we have a bound \bar{N} on N , then a sensible value for ℓ_A is

$$\ell_A = \ell - \lceil \log 2(\bar{N}) \rceil . \quad (21.2)$$

This guarantees that the accumulation of errors due to possibly truncating input summands to ℓ_A will be smaller than 2^ℓ . Again, beware that this only bounds the error due to the accumulation itself: it is but one element of a complete error analysis (see Chap. 3).

An important message here is that a margin of three orders of magnitude (which is overwhelming by physics standards) corresponds to $\log 2(10^3) \approx 10$ bits added to the architecture. Section 21.2 will show that 10 more bits do not add much to the cost.

The three parameters m_X , m_A , and ℓ_A can be expressed as `#pragma` in an HLS context [Ugu+20], with the values given in Table 21.1 or Table 21.1 providing fallback defaults.

21.2 Accumulator Architectures

We now discuss the construction of an accurate or exact accumulator, whose main blocks are visible on Fig. 21.2. For a sum of product, the input to the accumulator comes from an exact multiplier, as shown by Fig. 21.6. The construction of this exact multiplier is described in Sect. 21.2.1. As it outputs its result in some floating-point format (albeit unnormalized), the construction of the accumulator itself (described in Sects. 21.2.2, 21.2.3, and 21.2.4) is the same for $\sum X_i$ and $\sum X_i Y_i$. If the result is needed in floating point, a fixed-to-float converter is described in Sect. 21.2.4.

Other potentially useful variations of Fig. 21.6 exist and will not be detailed further. For instance, it can be simplified into an operator for the sum of squares.

21.2.1 Exact Floating-Point Multiplier

Figure 21.7 gives the architecture of an exact, unnormalized multiplier for IEEEfloat format. It is rather simpler than the standard floating-point multiplier described in Sect. 11.2. Subnormals, in particular, are cheap to manage here: the “is normal” bit n_x is computed (by a wide OR) and prepended

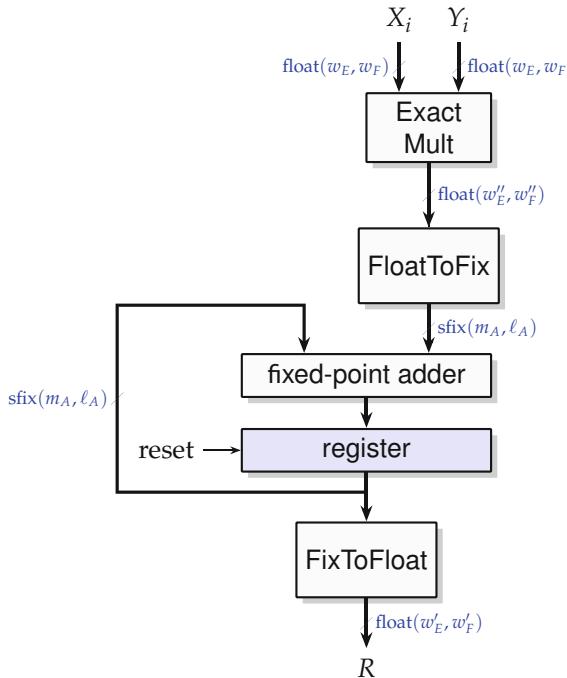


Fig. 21.6 Accurate or exact sum-of-product operator.

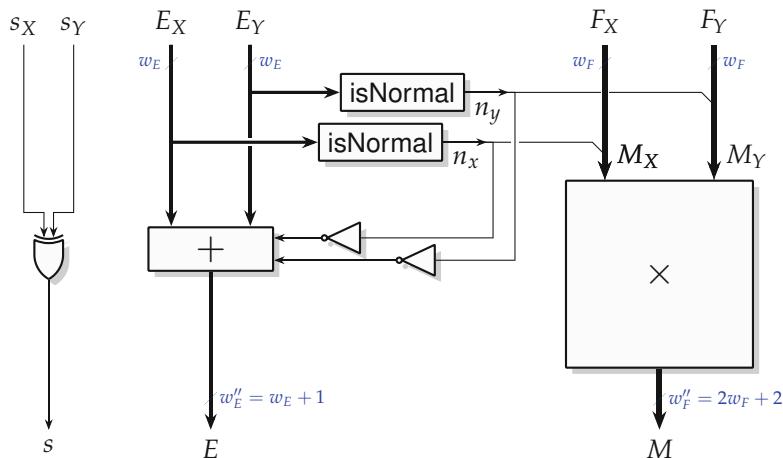


Fig. 21.7 Exact floating-point multiplier for an IEEE format.

to the fraction F_X to build $M_X = n_X + F_X$ and similarly for Y . The significand multiplier provides an exact result $M = M_X \times M_Y$ which is neither normalized nor rounded. Its true exponent is $E_X + (1 - n_X) - E_0 + E_Y + (1 - n_Y) - E_0$, where E_0 is the standard exponent bias, but it is both cheaper and more convenient to compute a doubly biased exponent $E = E_X + (1 - n_X) + E_Y + (1 - n_Y)$: this is exactly the number of bits by which the significand M should be left-shifted in order to align it to an exact accumulator (whose LSB is $2e_{\min} - 2w_F$; see Table 21.1).

If we use an application-specific value of $\ell_A > 2e_{\min} - 2w_F$, then the shift distance will be obtained by subtracting the (constant) difference $\ell_A - (2e_{\min} - 2w_F)$ to E . As ℓ_A is a parameter of the accumulator, not the multiplier, we consider for clarity that this subtraction belongs to the accumulator: it is not shown in Fig. 21.7. However, it should be clear that it can be computed in parallel with the product.

An exact multiplier for an Nfloat format is even simpler, as it does not require the `isNormal` boxes.

To sum up, compared to a standard floating-point multiplier which requires rounding and normalization logic including a costly shifter in the case of subnormal input (see Sect. 11.2.2, p. 345), the above exact multiplier is simpler, faster, and cheaper. Note, however, that its output is larger: this will entail some overhead in the following accumulator.

The next sections study the accumulation of floating-point values that may be either standard floating-point numbers with parameters w_E and w_F or exact products which are technically also floating-point numbers with parameters $w''_E = w_E + 1$ and $w''_F = 2w_F + 2$ (see Fig. 21.7). To cover these two cases, the accumulator architectures are described in terms of w''_E and w''_F , and the case of a simple accumulation corresponds to $w''_E = w_E$ and $w''_F = w_F$. In both cases, the inputs to the accumulator are a sign s , an exponent E , and an unnormalized mantissa M .

21.2.2 Simple Accumulator Architectures for Small Precisions

21.2.2.1 Exact Accumulator

The architecture of Fig. 21.8 was introduced in [Din+08]. The accumulator is kept in two's complement. A negative summand is converted in two's complement after the shift using (1) an array of 2-input XOR gates and (2) a carry-in to the adder. Together, they implement the well-known equation $-x = \bar{x} + 1$.

Figure 21.9 shows a slight improvement where the XOR is performed before the significand shift, leading to a smaller XOR. This requires a modified shifter that pads left and right with the sign bit s instead of padding with zeros (see Chap. 10 for shifter architectures). This, however, is mostly for free:

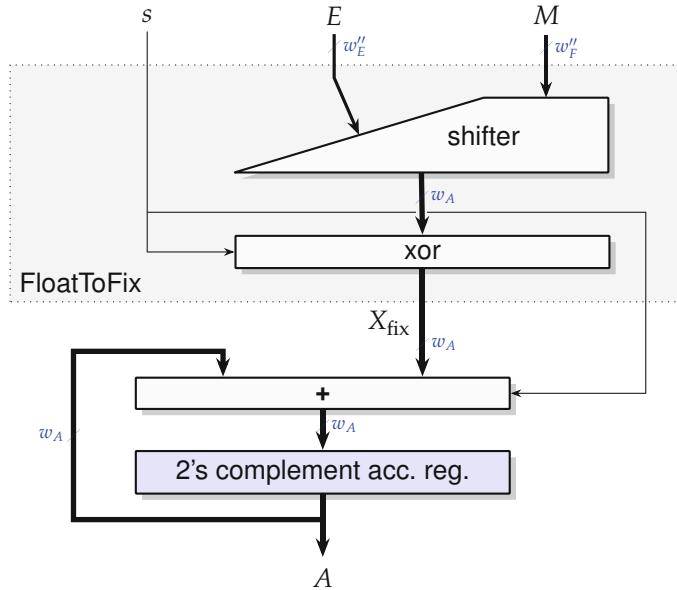


Fig. 21.8 Simple accumulator in two's complement fixed point.

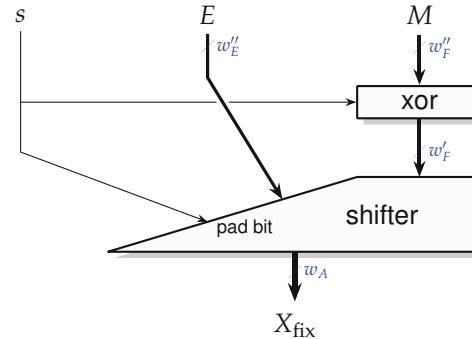


Fig. 21.9 Alternative Float2Fix architecture with smaller XOR.

the only impact would be the fan-out on \$s\$, but this fan-out remains close to \$w_A\$, as can be observed by counting the padding bits in the shift architectures of Chap. 10. It was also \$w_A\$ in the wider XOR array of Fig. 21.8.

21.2.2.2 Exception Management

A floating-point accumulator has to handle the cases when an input is an infinity or a NaN. They are managed by a very simple Finite State Machine

(FSM) with four states: finite (initial state, at a reset of the accumulator), $+\infty$, $-\infty$, and NaN. This FSM implements in the accumulator the standard behavior of a floating-point adder on exceptional inputs (see Table 11.1, p. 331).

21.2.2.3 Application-Specific Accumulator

The two previous figures describe a `FloatToFix` component for the exact accumulation case: no overflow nor rounding can happen during the accumulation. Conversely, a smaller accumulator tailored for an application must manage the case when some of its inputs will overflow or underflow. The overflow is detected by a comparison of the input exponent E with the constant m_A (corrected with the bias). Overflow (and their sign) are input to the FSM that manage exceptions. The shift distance is also computed by subtracting a constant to the input exponent E .

Concerning underflows, they can be either truncated or rounded. For truncation, the `FloatToFix` block simply uses a truncated shifter (see Chap. 10). If the choice is to round to nearest the underflowing inputs, the `FloatToFix` block is still a truncated shifter, but it produces a number in the $\text{sfix}(m_X, \ell_A - 1)$ format. The adder is enlarged by one bit to add a constant LSB of 1 (rounding bit). The LSB of the adder output is then dropped, and the accumulator register remains w_A -bits wide.

21.2.2.4 Two's Complement Versus Sign-Magnitude

An alternative would be to keep the accumulator in sign/magnitude representation, just like floating-point numbers. In this case, an input summand would need to be either added or subtracted to the accumulator, depending on the respective signs of the input and the accumulator. Then, when the accumulator becomes negative, its opposite would need to be computed, which would require another carry propagation. To avoid having two w_A -bit adders on the critical path of the loop, both $A + X_{\text{fix}}$ and $A - X_{\text{fix}}$ could be computed in parallel (possibly using the compound adder of Sect. 5.3.7, p. 123). These solutions have been studied in detail [UD17], and they involve more hardware than a two's complement accumulator as on Fig. 21.8 or Fig. 21.9.

Of course, in the big picture of Fig. 21.2, a two's complement accumulator will eventually also have to be converted to sign-magnitude in the `Fix to Float` box. However, this conversion can then be performed after the normalization, hence on w_F bits instead of w_A . The two's complement accumulator is therefore overall cheaper. Nevertheless, its main advantage is to simplify the critical path on the loop: this allows for a higher-frequency accumulation.

On modern FPGAs, a two's complement accumulator based on plain carry-propagate addition is very compact (an FPGA logic cell provides a bit of register behind each bit of addition) and uses only local routing resources and fast carry logic. For instance, a 64-bit accumulator achieves a frequency above 200 MHz.

We now review a simple technique that allows to achieve higher frequencies on wider accumulators, still using only plain carry-propagate adders.

21.2.3 High-Radix Carry-Save Architecture

To increase frequency, it is possible to pipeline arbitrarily the exact multiplier, the [FixToFloat] block, and the [FixToFloat] block from Fig. 21.6: they are combinatorial. However, the w_A -bit addition cannot be pipelined since it is on a loop.

To achieve a high frequency for this w_A -bit addition of the accumulator, the cheapest option [MRR91; Din+08; UD17] is to simply register the carry propagation every α bits as shown in Fig. 21.10. Both the input X_{fix} and the register output A are split into $k = \lceil w_A / \alpha \rceil$ chunks of α bits (see also Sect. 2.3). The w_A -bit adder is also split into k smaller α -bit adders. At each cycle, all the α -bit adders compute in parallel the sum of their chunk of X , their chunk of A , and a carry-in from the previous chunk at the previous cycle. The addition of an input X_{fix} is thus only complete after k cycles, but one new value can be input at each cycle. The accumulated sum is still stored exactly, but in a radix- 2^α carry-save representation (introduced in Sect. 5.2.6, p. 112).

This solution only costs $k - 1$ additional registers (the c_i in Fig. 21.10).

This architecture enables single-cycle accumulation at arbitrary frequencies: the smaller α , the higher the frequency. On FPGAs with fast carry chains, $\alpha = 32$ already allows to reach a very high frequency, again thanks to

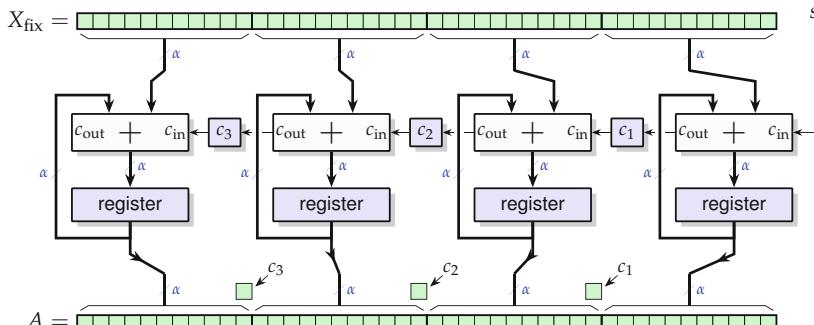


Fig. 21.10 High-radix carry-save accumulator.

the fact that the loops of Fig. 21.10 only involve local routing: the overhead of this solution, compared to the large ripple-carry adder of Fig. 21.8, is only a few percent.

Note that the carry register bits are not output in Fig. 21.10. To recover the actual value of the accumulator, it will be necessary to first complete the carry propagation. This can be achieved by inputting $X_{\text{fix}} = 0$ for k cycles at the end of the computation (3 cycles in Fig. 21.10). These few cycles are easily amortized for large computations.

If the running sum is needed at each cycle, then the c_i must be output to the **FixToFloat** block, where a separate adder will propagate them (again in k cycles but in a pipelined way, without requiring a zero input for k consecutive cycles). This will be detailed in Sect. 21.2.4. There will be some overhead but, again, out of the loop critical path.

21.2.4 Conversion of the Accumulator Back to Floating Point

We now discuss the **FixToFloat** block that converts the accumulator value back to floating point. This conversion is parameterized by w_A , possibly the chunk size α and the parameters (w'_E, w'_F) of the output format.

Let us first remark, using a few examples, that this component is probably much less useful than the accumulator itself in an application-specific context.

In the inductance computation example [Cre+07], the FPGA computes a very large integration – several hours – and only the final result is relevant. In such applications, it makes no sense to dedicate hardware to the conversion of the accumulator back to floating point. FPGA resources will be better exploited at speeding up the computation as much as possible, and the conversion is best performed in software.

Another common case is that one needs one normalization every N accumulations. For instance, a dot product of vectors of size N accumulates N products before needing to convert the result back to floating point. Therefore, in matrix operations, one pipelined **FixToFloat** operator may be shared between up to N dot product operators [ZP05], at the cost of some multiplexers and routing. Experience has shown that such sharing is well exploited by HLS compilers [UD17; Ugu+20].

It therefore makes sense to provide **FixToFloat** as a separate component, as in Fig. 21.6.

Let us first discuss the simpler case when the accumulator is within the target exponent range $[e_{\min}, e_{\max}]$ of the output format: it covers the exact sum operator and most application-specific accumulators. Figure 21.11 shows the components needed for this conversion. The most expensive component is the **normalizer**, a combined leading sign counter and shifter (see Sect. 10.3, p. 323). It outputs the normalized (but still signed) fraction F_s

(without the leading bit, but with a rounding bit at the LSB) and the leading sign bit count C .

This tentative fraction is then bitwise negated if $s = 1$ thanks to a row of $w'_F + 1$ XOR gates. Meanwhile, the tentative biased exponent E is computed as $E_0 + m_A - C$ in the [exponent processing] block. This box also produces a zero signal z when $C = w_A$.

The [rounding, exception handling & pack] block concatenates E and F and then adds at the LSB the sum of the sign bit s (to complete the negation started by the [xor] block) and the rounding bit. This requires an adder of $w'_F + w'_E$ bits. Thanks to the clever encoding of floating-point numbers (see Table 2.1, p. 51), the possible fraction overflow in this addition translates to an increase of the exponent field. This captures overflows due to rounding, but also the overflow due to the extreme negative value -2^{m_A} of the accumulator.

The final exception status is determined out of the result of the previous adder, the exception bits from the accumulator, and the zero bit z . The details and the encoding of these exceptions depend on the format used for the output, e.g., IEEEfloat or Nfloat: they is left as an exercise for the reader.

The discussion so far addresses rounding to nearest with ties to away. There is no standard here imposing the *ties to even* rule, but it may be desirable if the output is IEEEfloat. It has some overhead, though, as the normalizer must now compute a sticky bit which is the OR of all the bits it discards (see Chap. 10). This sticky bit is then used in the rounding logic just like in a floating-point adder.

The case when the accumulator is wider than the target exponent range $[e_{\min}, e_{\max}]$ is quite similar. The main difference is that some bits of an exact accumulator can never make it to the floating-point result in the output

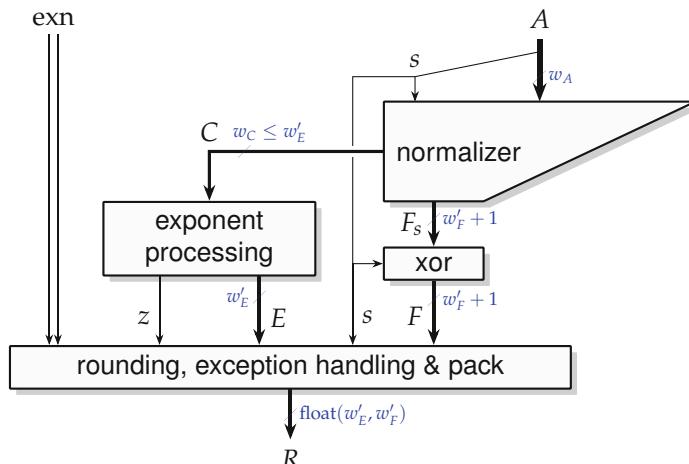


Fig. 21.11 Conversion of the large accumulator to floating point.

format (w'_E, w'_F) . This is illustrated by Fig. 21.12 for the worst case (in terms of accumulator size), which is the exact sum of products. For the `FixToFloat` component, the main differences with Fig. 21.11 are the following:

- The input to the normalizer is not the full accumulator, but only the bits in the $[e_{\min} - w_F, e_{\max}]$ range. However, the normalizer counts and shifts no more than $e_{\max} - e_{\min}$ bits. There is no special treatment to do for subnormals, since all the subnormals are fixed-point numbers with the e_{\min} exponent.
- Positive and negative overflows are detected out of the bits higher than e_{\max} : if they are not all identical to the sign bit, there is an overflow.
- For the *ties to even* rule, a sticky bit must be computed out of the bits lower than $e_{\min} - w_F$ and ORed with the sticky bit computed by the normalizer.

21.3 Cost, Speed, and Accuracy

By comparing the architecture of a floating-point adder and that of a large accumulator with its `FixToFloat` conversion, one may derive a few rules of thumb, which have been supported by experience [Din+08; UD17; Ugu+20]:

- The exact multiplier is slightly cheaper and faster than an Nfloat multiplier and much cheaper than an IEEEfloat one (as the latter must manage subnormals).
- The combination of `FloatToFix`, fixed-point accumulator, and `FixToFloat` components is comparable in size to a floating-point adder with fraction size w_A . For exact accumulators, this is huge, for instance, an exact accumulator for 32-bit floating point is about ten times the size of a 32-bit floating-point adder [UD17]. However, application-tailored accumulators are very competitive: for 32-bit data, a 64-bit accurate accumulator is comparable in size to a 64-bit floating-point adder (and may be much smaller if $m_X < \text{MSBA}$) while accepting a new input at each cycle and offering better accuracy [Din+08; Ugu+20].
- The fixed-point adder and its registers are much smaller than both `FloatToFix` and `FixToFloat`.

These are but rules of thumb that should be confirmed by application-specific parameterization and measures, in particular for accuracy. How-

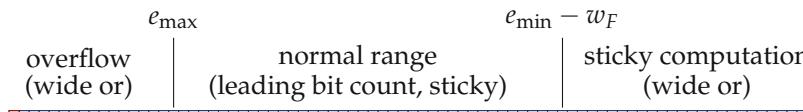


Fig. 21.12 Structure of the accumulator for an exact IEEE 754 sum-of-product operator.

ever, they reinforce a previously stated message: adding 10 bits here or there may dramatically improve confidence in the parameterization and will not dramatically impact the cost of the architecture.

21.4 To Probe Further

There have been many implementations of exact accumulators [KK88; Kno91; MRR91; Ker+94; Kul13; NB14; Koe+17; Bru17; FN18]. The present chapter focused on the simplest and most cost-effective solution after a comparison of these architectures in terms of cost and performance [UD17].

There are several trivial simplifications to perform when the summands are all positive. A sum of square accumulator can, in addition, benefit from squarer improvements presented in Chap. 14.

References

- [Bod+06] Michael R. Bodnar, John R. Humphrey, Petersen F. Curt, James P. Durbano, and Dennis W. Prather. “Floating-Point Accumulation Circuit for Matrix Applications”. In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2006, pp. 303–304. (cit. on p. [624](#)).
- [Bru17] Nicolas Brunie. “Modified Fused Multiply and Add for exact low precision product accumulation”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2017, pp. 106–113. (cit. on p. [639](#)).
- [CM88] Peter R. Cappello and Willard L. Miranker. “Systolic super summation”. In: *Transactions on Computers* 37.6 (1988), pp. 657–677. (cit. on p. [627](#)).
- [Cre+07] Octavian Creț, Ionuț Trestian, Radu Tudoran, Laura Darabant, Lucia Văcariu, and Florent de Dinechin. “Accelerating The Computation of The Physical Parameters Involved in Transcranial Magnetic Stimulation Using FPGA Devices.” In: *Romanian Journal of Information, Science and Technology* 10.4 (2007), pp. 361–379. (cit. on pp. [629](#), [636](#)).
- [Din+08] Florent de Dinechin, Bogdan Pasca, Octavian Creț, and Radu Tudoran. “An FPGA-specific Approach to Floating-Point Accumulation and Sum-of-Products”. In: *Field-Programmable Technologies (FPT)*. IEEE, 2008, pp. 33–40. (cit. on pp. [632](#), [635](#), [638](#)).
- [FN18] Luiś Fiolhais and Horácio Neto. “An Efficient Exact Fused Dot Product Processor in FPGA”. In: *International Conference on Field*

- Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 327–3273. (cit. on p. 639).
- [Ker+94] Juergen Kernhof, Christoph Baumhof, Bernd Höfflinger, Ulrich Kulisch, Steve Kwee, Peter Schramm, Manfred Selzer, and Thomas Teufel. “A CMOS floating-point processing chip for verified exact vector arithmetic”. In: *European Solid-State Circuits Conference*. IEEE, 1994, pp. 196–199. (cit. on pp. 627, 639).
- [KK88] Reinhard Kirchner and Ulrich Kulisch. “Accurate arithmetic for vector processors”. In: *Journal of Parallel and Distributed Computing* 5.3 (1988), pp. 250–270. (cit. on pp. 627, 639).
- [Kno91] Andreas Knoefel. “Fast Hardware Units for the Computation of Accurate Dot Products”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 1991, pp. 70–74. (cit. on p. 639).
- [Koe+17] Jack Koenig, David Biancolin, Jonathan Bachrach, and Krste Asanovic. “A hardware accelerator for computing an exact dot product”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2017, pp. 114–121. (cit. on p. 639).
- [Kul13] Ulrich Kulisch. *Computer arithmetic and validity: theory, implementation, and applications*. Walter de Gruyter, 2013. (cit. on pp. 627, 639).
- [Lut19] David R. Lutz. “ARM Floating-Point 2019: Latency, Area, Power”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2019, pp. 69–76. (cit. on p. 624).
- [MRR91] Michael Müller, Christine Rüb, and Wolfgang Rülling. “Exact accumulation of floating-point numbers”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 1991, pp. 64–69. (cit. on pp. 635, 639).
- [Mul+18] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. 2nd ed. Birkhäuser Boston, 2018. (cit. on p. 624).
- [NB14] Fabian Nowak and Rainer Buchty. “A Tightly Coupled Accelerator Infrastructure for Exact Arithmetics”. In: *Architectures of Computing Systems*. LNCS 5974. 2014. (cit. on p. 639).
- [UD17] Yohann Uguen and Florent de Dinechin. “Design-space exploration for the Kulisch accumulator”. 2017. URL: <https://hal.archives-ouvertes.fr/hal-01488916>. (cit. on pp. 634, 635, 636, 638, 639).
- [Ugu+20] Yohann Uguen, Florent de Dinechin, Victor Lezaud, and Steven Derrien. “Application-Specific Arithmetic in High-Level Synthesis Tools”. In: *ACM Transactions on Architecture and Code Optimization* 17.1 (2020). (cit. on pp. 628, 630, 636, 638).
- [ZP05] Ling Zhuo and Viktor K. Prasanna. “High Performance Linear Algebra Operations on Reconfigurable Systems”. In: *Supercomputing*. IEEE, 2005. (cit. on pp. 624, 636).



CHAPTER 22

Floating-Point Exponential

It can't continue forever. The nature of exponentials is that you push them out and eventually disaster happens.

Gordon Moore

The exponential function is, after the basic arithmetic operators, one of the next most useful building blocks in scientific computing. This chapter first reviews some of the techniques that can be used to evaluate it. It then presents in detail an implementation of a complete floating-point exponential operator. This chapter is also a good example of the composition of many arithmetic components in a coarser operator computing just right.

The exponential function is among the most used elementary functions of the standard mathematical library. For this reason, it is provided in hardware form in some graphics processing units (GPUs) [OS05] and in neural network acceleration circuits [Mik+18]. On FPGAs, it has been used for scientific and financial Monte Carlo simulations [Ech+08; Jin+12], to accelerate the SPICE electrical circuit simulator [KD09], in Bayesian network learning [HPS13], in biologically accurate neuron simulation [MMN13], in kernel adaptive filters [Fra+17], and in the implementation of the power function [EL08; Din+13] among others.

22.1 Mathematical Background

The exponential of a real number is the function e^x , where $e \approx 2.71828183\dots$ is Euler's number. Its computer incarnation is usually denoted as $\exp(x)$. It is defined on the set of real numbers, and its output is a strictly positive real. It is plotted in Fig. 22.1.

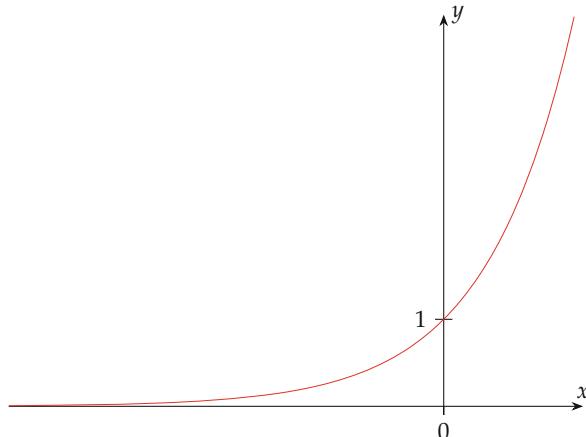


Fig. 22.1 Plot of the exponential function.

Among the many identities involving the exponential [AS64], here are the three which are the most useful to its implementation:

$$e^{x+y} = e^x \times e^y \quad \forall x, y \in \mathbb{R}, \quad (22.1)$$

$$2^x = e^{x \log 2} \quad \forall x \in \mathbb{R}, \quad (22.2)$$

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \dots \quad \forall x \in \mathbb{R}. \quad (22.3)$$

We will now look into the specifics when evaluating the exponential function using specific number formats.

22.2 Number Formats and the Exponential Function

22.2.1 Fixed Point Versus Floating Point

It is well known that the exponential function grows extremely fast. A consequence is that an exponential function with identical input and output formats is in general a mismatch.

For instance, if the input x is a signed fixed-point format $\text{sfix}(m, \ell)$ with $m > 0$, the output may reach $e^{2^m - 1} = 2^{\frac{2^m - 1}{\log 2}}$. In other words, its MSB should be $m' = \left\lceil \frac{2^m - 1}{\log 2} \right\rceil \approx \lceil 1.4427(2^m - 1) \rceil$, which is much larger than m as soon

as $m > 1$. For instance, for $m = 7$ (a format with 8 integer bits), the MSB of a fixed-point exponential should be $m' = 184$.

For this reason, the exponential function will be used for fixed-point numbers with very few integer bits, if any.

This is also the reason why the exponential function is mostly used with floating-point formats, and the rest of this chapter addresses the floating-point exponential. In this case, the output makes full use of the format, and the issue is now that the input format is underused: for large positive values, the exponential will overflow (return $+\infty$), while for large negative values, it will underflow (return $+0$). This will be more formally defined below.

22.2.2 Fixed-Point-In, Floating-Point-Out Version

From the considerations above, the natural definition of an exponential computing just right would be that it inputs a fixed-point format and outputs a floating-point format. Such an exponential is not a standard one but is easy to build; it is actually a sub-circuit of the floating-point operator designed in the sequel.

This has an important practical consequence: if an application computes the exponential of a sum, especially a sum of many components, then its hardware implementation should not compute this sum in floating point. Instead, it should perform the sum in fixed point and then input it to a fixed-point-in, floating-point-out exponential. This will be more efficient, but also more accurate, since a fixed-point summation is exact as long as no overflow occurs.

A symmetrical discussion applies to the logarithm function, whose “natural” operator should input a floating-point format and output a fixed-point one [Le+16]. If an application computes a sum of logarithm, then using a floating-point-in, fixed-point out logarithm and performing the sum in fixed point will be more efficient and accurate than performing the sum in floating point.

22.2.3 Input Filtering

Back to a standard, floating-point-in, floating-point-out exponential function, Table 22.1 uses the extremal values of the floating-point formats (from Table 2.4) to define the underflow threshold X_{\min} and the overflow threshold X_{\max} . The exponential will return zero for all input numbers $X < X_{\min}$ and will return $+\infty$ for all input numbers $X > X_{\max}$. In single precision, for instance, the set of input numbers on which $\exp(X)$ is finite is approximately $[-103.3, 88.73]$ in IEEEfloat(8,23) and $[-88.03, 89.42]$ in

Table 22.1 Borderline cases for floating-point exponential. The rounding up and down are to the respective floating-point format.

	IEEEfloat(w_E, w_F)	Nfloat(w_E, w_F)
X_{\min}	$-\left\lfloor \log\left(2^{-2^{w_E-1}} + 2^{-w_F}\right) \right\rfloor$	$-\left\lfloor \log\left(2^{-2^{w_E-1}} + 1\right) \right\rfloor$
X_{\max}	$\left\lceil \log\left((2 - 2^{-w_F}) \cdot 2^{2^{w_E-1}-1}\right) \right\rceil$	$\left\lceil \log\left((2 - 2^{-w_F}) \cdot 2^{2^{w_E-1}}\right) \right\rceil$
X_1	2^{-w_F-2}	2^{-w_F-2}

Nfloat(8, 23). Note how the existence of subnormals extends the underflow threshold for IEEEfloat.

These values do not need to be hardcoded in the architecture, though: it may be more economical to implement a first very coarse overflow/underflow detection (for instance, underflow if exponent is smaller than $-w_E + 1$ and overflow if exponent is larger than $w_E - 1$) and then add underflow/overflow detection logic to the main datapath.

When X is small in absolute value, consider the power series (22.3). As soon as $|X| \leq X_1 = 2^{-w_F-2}$, we have $e^X - 1 < 2^{-w_F-1}$, and 2^{-w_F-1} is the half-ulp of 1.0: the exponential will return 1.0.

These special cases are summarized in Fig. 22.2.

One consequence of this analysis is that the test of a floating-point exponential operator should focus on numbers between X_{\min} and X_{\max} . In FloPoCo's test bench generator for FPExp, the exponent of the random inputs is restricted to $[-w_F - 3, w_E - 2]$.

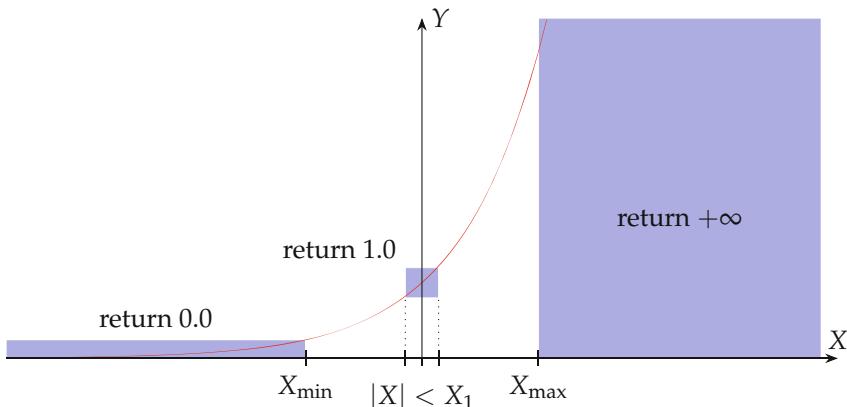


Fig. 22.2 Special cases for the exponential function.

22.2.4 A First Architecture: Direct Tabulation

Among the binary floating-point representations, the two intervals $[X_{\min}, -X_1]$ and $[X_1, X_{\max}]$ have a very small exponent range: the exponent of X_1 is $E_{X_1} = -w_F - 2$, and the exponent of both X_{\min} and X_{\max} is $w_E - 2$. We therefore have an exponent amplitude of $w_E + w_F + 1$ in both cases: tabulating all the finite values of the exponential will require two tables of $2^{\log_2(w_E+w_F+1)+w_F}$ entries each.

For the very small formats used for machine learning such as BFloat, DLFLOAT, and MSFP8-11 (see Sect. 2.5), this is a relevant option. For instance, BFloat/IEEEfloat(8,7) requires two tables of $2^{4+7} \times 15$ bits (the output is 15 bits only since there is no need to store the constant sign).

Around these tables, all that is needed is some overflow/underflow management logic and a small subtraction on the exponent field to compute the higher bits of the table input as $E_X - E_0 - E_{X_1}$. The lower bits are simply the fraction bits. The sign bit can be concatenated to have a single table, as illustrated in Fig. 22.3.

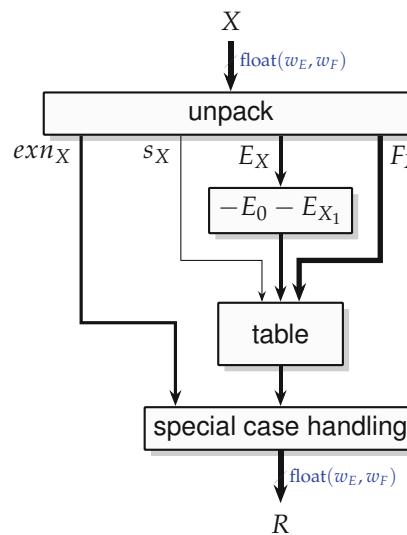


Fig. 22.3 Direct tabulation architecture for small precisions.

A natural idea, to address larger precisions, is then to split X into a high part X_H (containing the sign, the exponent, and the upper bits of the fraction) and a low part X_L (containing the remaining fraction bits). This split can be written $X = X_H + X_L$ where both X_H and X_L are floating-point numbers. The exponential is then $e^X = e^{X_H} \times e^{X_L}$. Both terms can be tabulated (the table for e^{X_H} with the same offset as in the direct tabulation above).

Actually, X_L is a fixed-point number smaller than 1, so e^{X_L} can be evaluated using, for example, a Taylor series instead. The reconstruction requires a multiplier of at least $w_F \times w_F$ bits (a few guard bits must be added). It is also possible to decompose X in more than two chunks, at the cost of more multipliers.

All this works, but we do not detail further this option here, because the range reduction in the following section, although more complex, scales better to large precisions and eventually has smaller hardware requirements.

22.3 Architecture for Floating-Point Range Reduction

The algorithm used is similar to the state of the art that is typically used in software [Tan89; Mar00; Li+01], but a hardware implementation exposes new trade-offs.

22.3.1 Overview

The main idea is to reduce X to an integer E and a fixed-point number Y , such that

$$X = E \cdot \log 2 + Y \quad (22.4)$$

To achieve this, the hardware first computes

$$E = \left\lfloor \frac{X}{\log 2} \right\rfloor \quad (22.5)$$

with a rounding to the nearest integer (no tie problem here as $\log 2$ is irrational). Then, the reduced argument Y is computed with

$$Y = X - E \times \log 2 \quad (22.6)$$

and it will belong to the interval $Y \in [-\frac{\log 2}{2}, \frac{\log 2}{2}]$.

As (22.4) can be rewritten

$$e^X = 2^E \cdot e^Y, \quad (22.7)$$

E is the exponent of the result, and e^Y is almost its significand. Indeed, with $Y \in [-\frac{\log 2}{2}, \frac{\log 2}{2}]$, we have $e^Y \in [0.7, 1.42]$, and a significand must be $1.F \in [1, 2)$. Thus, the exponent and significand of the result may be obtained as

$$R = \begin{cases} 2^E \cdot e^Y & \text{if } e^Y \geq 1 \\ 2^{E-1} \cdot (2e^Y) & \text{if } e^Y < 1. \end{cases} \quad (22.8)$$

This test is implemented by a multiplexer controlled by the most significant bit of e^Y , and the multiplication by 2 is just a wired shift.

The architecture of an exponential operator using this range reduction is given in Fig. 22.4. It is slightly complicated by the fact that the significand of X is provided in sign (s_X) and magnitude ($1.F_X$). Therefore, E is also represented as its absolute value $|E|$, and it inherits the sign s_X .

All the previous assumes perfectly accurate computations. Let us now see how they can be relaxed and approximated in a practical architecture.

22.3.2 Accuracy Considerations

The hardware for (22.5) is essentially a constant multiplier (see Chap. 12). However, computing this value perfectly accurately will be quite expensive, requiring the irrational constant $1/\log 2$ to be evaluated at least to the precision of X . Fortunately, there is no need to compute (22.5) perfectly. Indeed, it may even be computed quite inaccurately, as long as (22.6) is computed accurately enough so that $2^E \cdot e^Y$ is an accurate representation of e^X . The fact that E is not the proper exponent for the normalized representation of e^X will be managed anyway by the normalization step (22.8).

In more details, an inaccurate computation of (22.5) will mainly have two consequences: (1) the value of E may be different from the ideal value, and (2) the interval of Y will be larger than $[-\frac{\log 2}{2}, \frac{\log 2}{2}]$.

From an architectural point of view, we can afford to enlarge a bit the interval of Y . In particular, enlarging each bound to the next power of two will not enlarge the bit vector holding Y . Furthermore, in the sequel, we will input the higher bits of Y to a table: it will be more architecture-friendly to round the number of table entries to a power of two. With an ideal implementation of (22.5), $\log 2/2 \approx 0.347$; the next power of 2 is 1/2, so only about 70% of the table addresses would be used.

For these two reasons, it is more hardware-efficient to use a cheaper, relaxed implementation of (22.5) that will just ensure $Y \in [-1/2, 1/2]$. Then we have $e^Y \in [0.6, 1.7]$: this is still in the working range of the normalization step (22.8).

Our astute reader probably wondered, when first reading (22.5), why we did not use instead the computation $E = \left\lfloor \frac{X}{\log 2} \right\rfloor$ (notice the floor operation) which would directly have provided the normalized representation: $Y \in [0, \log 2]$, entailing that $2^Y \in [1, 2]$ is a properly normalized significand, without the need of the correction step (22.8). The answer is that this would only work with a correctly rounded implementation of the whole compu-

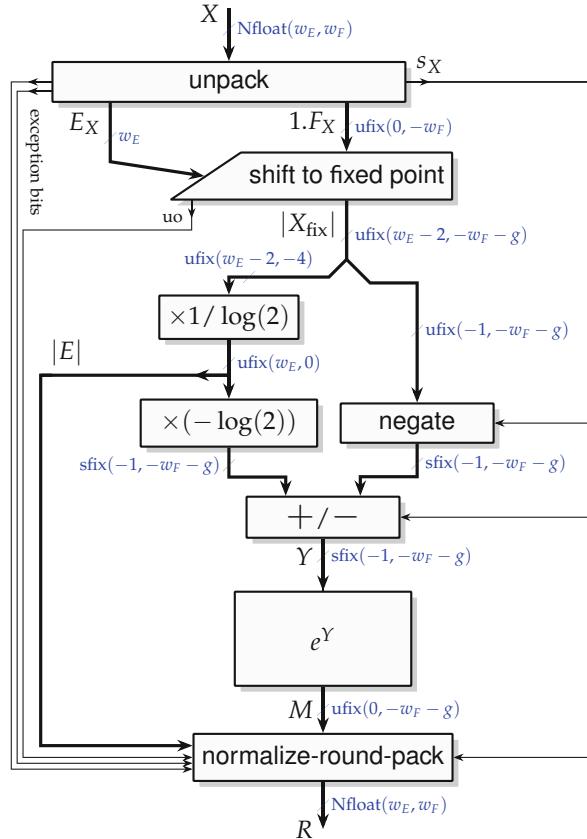


Fig. 22.4 Architecture of FPExp for the Nfloat(w_E, w_F) format.

tation (this constant multiplication but also the rest of the datapath). This would be extremely expensive to implement, in particular requiring very high internal accuracies [Mul+18]. As soon as the target is a faithful architecture, the 1-bit possible error entails that a final correction step is needed. It then becomes much more efficient to exploit it to relax the whole computation, as we suggested above.

22.3.3 Fixed-Point Conversion

As (22.5) and (22.6) are inherently fixed-point computations, the first task is to build a fixed-point representation X_{fix} of the input X . The most significant bit (MSB) of this representation is provided by the conditions $X > X_{\max} \Rightarrow \exp(X) = +\infty$ and $X < X_{\min} \Rightarrow \exp(X) = 0$, which can be relaxed to

$$|X| \geq 2^{w_E-1} \Rightarrow \exp(X) = 0 \text{ or } +\infty . \quad (22.9)$$

For instance, in IEEEfloat(8, 23), we have $X_{\min} = -103.3$ and $X_{\max} = 88.73$, so any X with $|X| \geq 2^7$ will lead to $\exp(X) = 0$ or $+\infty$.

As the floating-point input is in sign/magnitude, this condition will simply be a test on the exponent, providing the uo (underflow/overflow) flag.

After this, the main datapath may focus on $|X| < 2^{w_E-1}$ which means that the MSB of the (unsigned) X_{fix} will be $w_E - 2$.

The least significant bit position of X_{fix} is defined by the fact that if $|X| < 2^{-w_F-2}$, then $\exp(X) = 1$ (see Fig. 22.2 and Table 22.1). This defines an LSB of weight $-w_F - 2$. However, more LSB bits will be needed to allow for a last-bit accurate result in the presence of approximation and rounding errors: let us define the LSB of X_{fix} as $-w_F - g$, for some number of guard bits $g \geq 2$. The error analysis that computes g is detailed below in Sect. 22.3.8.

The [shift to fixed point] box in Fig. 22.4 shifts the significand by the value of the exponent. More specifically, if the exponent is positive, it shifts to the left by up to $w_E - 2$ positions (more means overflow); if the exponent is negative, it shifts to the right by up to $w_F + g$ positions. As suggested in Chap. 10, both shift cases are best implemented by a single left shifter, the shift distance being $E_X - (E_0 + w_F + g)$ where E_X is the exponent field of X and E_0 is the floating-point exponent bias (see Sect. 2.5). The output of this shifter is $|X_{\text{fix}}|$ in the ufix($w_E - 2, -w_F - g$) format.

The computation of the shift distance is a single small subtraction, $E_0 + w_F + g$ being a constant. If this shift distance is negative, X_{fix} is forced to 0. If it is larger than $w_E - 2 + w_F + g$, the uo signal is produced, and the value of $|X_{\text{fix}}|$ becomes irrelevant.

This box is no more complex in the IEEEfloat case, as all input subnormals belong to the “return 1.0” region: there is no need for a specific management of input subnormals.

22.3.4 Computation of the Tentative Exponent

Let us now turn to the relaxed computation of the tentative exponent E (see Fig. 22.4). According to (22.8), E is equal either to E_R or to $E_R + 1$, where E_R is the correct exponent of the final result, on w_E bits.

E must be a signed integer on $w_E + 1$ bits, the extra bit being needed to accommodate the possible overflow in the $E_R + 1$ case when $E_R = e_{\max}$. Note that we will also have such an overflow for all the overflowing input values that have not been captured by the uo bit, e.g., inputs between 88.73 and 127 for IEEEfloat(8, 23). The whole datapath must be dimensioned such that these situations can be detected in the [normalize-round-pack] box.

Let us now determine the error we are allowed to perform in the relaxed computation of E . The constraint here is that the corresponding value of Y

should be such that $Y \in \left[-\frac{1}{2}, \frac{1}{2}\right)$. In the worst case, $Y = -1/2$, the computed value of $E \log 2$ is off by $1/2 - \log 2$. This means that the computation $\frac{X_{\text{fix}}}{\log 2}$, before rounding to an integer, was off by $\frac{1-\log 2}{2\log 2} \leq 0.22$. This is therefore the error bound that the architecture computing $\frac{X_{\text{fix}}}{\log 2}$ must ensure before rounding this product to an integer.

There are several ways to implement this operation. First, it will save hardware to truncate X_{fix} to some LSB ℓ_X . The obvious choice here is to truncate to $\ell_X = -4$, entailing an error bounded by 2^{-4} , which will be amplified by $1/\log(2)$ to $\bar{\delta}_{\text{trunc}} \approx 0.09$, or about half our error budget of 0.22. In FloPoCo, a FixRealKCM is invoked that will directly round to an integer. The FixRealKCM interface offers the possibility to specify an optional accuracy bound, which in the present case will be $0.5 + 0.13$. Here, 0.5 is a bound of the error of the final rounding to an integer, and $0.13 = 0.22 - 0.09$ is the overall error budget remaining for the multiplication of the truncated X_{fix} by the real $\log(2)$.

Another more generic option, since $X_{\text{fix}} < 2^{w_E-1}$, is to round the constant $1/\log 2$ to $\ell_C = -3 - w_E$, then use an exact multiplier, and round its result to the nearest.

22.3.5 Computation of Y

Let us now turn to the implementation of (22.6) as

$$Y = X_{\text{fix}} - E \times \log 2. \quad (22.10)$$

Here, we need another constant multiplier, this time by $\log 2$. It is very nonstandard:

- The subsequent fixed-point subtraction, by construction, will cancel the integer part and the first bit of the fractional part. For this reason, these bits do not even need to be computed by the constant multiplier.
- This multiplier has a few input bits ($w_E + 1$) and many output bits. This is a good case for a KCM multiplier (see Chap. 12).

Then, since at this point in the architecture we still have X_{fix} in (sign, magnitude) format, we need to compute $|X_{\text{fix}}| - E \log 2$ and then take its opposite if $s_X = 1$. The architecture shown in Fig. 22.4 is an alternative that computes either $|X_{\text{fix}}| + (-E \log 2)$ or $-|X_{\text{fix}}| - (-E \log 2)$, depending on s_X . It has a slightly better latency on FPGAs.

22.3.6 Computation of e^Y

Since $Y \in (-\frac{1}{2}, \frac{1}{2})$, we will have $e^Y \in [0.65, 1.65]$: its MSB is at bit position 0. Besides, since the interval of e^Y does not contain zero, it can be completely computed in fixed point without needing to worry about catastrophic cancellations.

There are several relevant ways of computing a fixed-point approximation M to e^Y , and they are reviewed in Sect. 22.5. They can be used in the architecture of Fig. 22.4 as soon as the implementation is accurate enough. The result must be output in a ufix($0, -w_f - g$) format such that its overall error $M - e^Y$ can be made arbitrarily small by increasing g . This constraint (in particular the constraint on g) will be formalized in the error analysis of Sect. 22.3.8.

22.3.7 Normalization and Rounding

A final normalization step implements (22.8): the test $e^Y \geq 1$ is actually a multiplexer controlled by the MSB of M . It possibly shifts left the significand by 1 bit. The final rounding then consists of possibly adding one half-ulp bit and then truncating. For this addition, the biased floating-point formats (see Sect. 2.5) have the nice property that a single adder of size $w_E + w_F + 1$ may be used to add the rounding bit to the concatenated exponent and significand: carry propagation from significand to exponent will handle the possible exponent change due to rounding up (see Table 2.1).

This is all what is needed for Nfloat formats. Unfortunately, for IEEEfloat, the result can become subnormal. The classic technique used in software is to delegate the normalization, rounding, and management of subnormal outputs to a floating-point multiplier computing $2^E \times M$. It would not be efficient here. Therefore, in this case, the `normalize-round-pack` box must include some exponent comparison logic and a shifter of w_F bits before the rounding adder.

22.3.8 Overall Error Analysis

The error analysis in this section may look much less formal than in the previous chapters, but the reader is invited to check that it simply shortcuts the technique introduced in Chap. 3 and used so far.

In the following, all the error bounds will be expressed in terms of unit in the last place (ulp) of Y , which has the value $u = 2^{-w_F-g}$. These error bounds can thus be made as small as required by increasing g .

The argument reduction is not exact. As already stated, numerical errors in the computation of E by (22.5) mostly impact the range of Y , not its accuracy. The accuracy of Y essentially depends on the computation of Y by (22.6). Here, we may distinguish two exclusive cases:

- If $|X|$ is small, more precisely if the exponent of X is smaller than -2 , then some of its LSBs may be lost in the right shift of $1.F_X$: this entails an error of at most one ulp in Y . However, in this case, (22.5) computes $E = 0$. This entails that the computation of $E \times \log 2$ will be exact (as soon as the constant multiplier is faithful or correctly rounded).
- Conversely, if X is large (its exponent is -1 or larger), then on the one hand, its significand is shifted left, hence without loss of information (this assumes $g \geq 1$ which will always be the case). On the other hand, the computation of $E \times \log 2$ introduces an error of at most one ulp, depending on the constant multiplication used.

In each of these cases, we thus have an error of at most one ulp on Y .

The **negate** box runs in parallel with the two constant multipliers: this will hide the latency of the carry propagation if a standard adder is used. An alternative is to replace it with a row of XORs to save hardware and power. This adds to Y another error bounded by one ulp when $X < 0$.

In summary, we have an error on Y that we can define as

$$\delta_Y = Y - (X - E \log(2)) \quad (22.11)$$

where E is the value used by the architecture (not the possibly different value that would be computed by an infinitely accurate multiplier by $1/\log(2)$). This error is bounded by some $\bar{\delta}_Y$ that depends on some of the architectural choices made and also on the interval of X . To fix ideas, we assume an exact **negate** box and a faithful constant multiplier, leading to $\bar{\delta}_Y = 1u$.

Let us now see how it propagates to e^Y . Again, the reference value is the ideal reduced argument for this value of E , so

$$\begin{aligned} \delta_M &= M - e^{X-E \log(2)} \\ &= (M - e^Y) + (e^Y - e^{X-E \log(2)}) \\ &= \delta_{\text{arch}} + \delta'_Y \end{aligned} \quad (22.12)$$

Here, $\delta_{\text{arch}} = M - e^Y$ is the evaluation error of the box computing e^Y : it depends on the technique used and will be bounded in Sect. 22.5. The term $\delta'_Y = e^Y - e^{X-E \log(2)}$ captures the amplification of δ_Y through the exponential. It can be rewritten

$$\begin{aligned}
 \delta'_Y &= e^Y - e^{Y-\delta_Y} \quad \text{using (22.11)} \\
 &= e^Y - e^Y e^{-\delta_Y} \\
 &= e^Y (1 - e^{-\delta_Y}) \\
 &= e^Y (\delta_Y - \delta_Y^2/2 + \delta_Y^3/6 \dots) \tag{22.13}
 \end{aligned}$$

$$\text{hence } |\delta'_Y| < e^Y |\delta_Y| (1 + u) \quad \text{if } \bar{\delta}_Y = 1u \tag{22.14}$$

Finally, the **normalize-round-pack** box implements (22.8) with a final rounding to the mantissa $1.F_R$ of the result. Here, we must distinguish the two cases of (22.8):

- If $M \geq 1$, then $1.F_R$ is M , rounded to position $-w_F$. The overall error decomposition is

$$\begin{aligned}
 \delta_R &= (1.F_R - M) + (M - e^{X-E \log(2)}) \\
 &= \delta_{\text{final round}} + \delta_{\text{arch}} + \delta'_Y \tag{22.15}
 \end{aligned}$$

In this case, we use in (22.14) the bound $1 \leq e^Y < 1.65$, entailing the bound

$$|\delta_R| < 2^{-w_F-1} + |\delta_{\text{arch}}| + 1.65(1+u)|\delta_Y| \tag{22.16}$$

- If $M < 1$, then $1.F_R$ is $2M$, rounded to position $-w_F$. The overall error decomposition is

$$\begin{aligned}
 \delta_R &= (1.F_R - 2M) + 2(M - e^{X-E \log(2)}) \\
 &= \delta_{\text{final round}} + 2\delta_{\text{arch}} + 2\delta'_Y \tag{22.17}
 \end{aligned}$$

However, in this case, we may use the bound $e^Y < 1$ in (22.14), entailing

$$|\delta_R| < 2^{-w_F-1} + 2|\delta_{\text{arch}}| + 2(1+u)|\delta_Y| \tag{22.18}$$

This latter case entails the largest error bound. If $\bar{\delta}_Y = 1u$, this encourages us to specify $\bar{\delta}_{\text{arch}} < 1 - 2u$ (a slightly-better-than-faithful e^Y architecture). Then, $\delta_R < 2^{-w_F-1} + 4u$. Since $u = 2^{-w_F-g}$, using $g = 3$ ensures the accuracy objective $\bar{\delta}_R < 2^{-w_F}$.

Note that for $g = 3$ we could even afford an error bound of $\bar{\delta}_{\text{arch}} = 2.3u$ in the evaluation of e^Y when $e^Y \geq 1$. It will be possible to exploit this in the sequel.

For IEEEfloat, the subnormal normalization shifter adds one ulp of error due to the bits shifted out, but in this case also divides $\bar{\delta}_M$ at least by 2; therefore, the overall bound of 4 ulps remain valid in this case.

Conversely, an approximate **negate** box as a row of XORs will cost us an increase of g to 4 (it adds one ulp only, but this ulp is amplified by e^Y), which probably costs more than it saves.

Finally, it is also possible to increase the parameter g beyond this minimal value of 3. More guard bits will mean a larger percentage of correctly rounded results. It is also useful when building larger faithful operator based on the exponential, in particular the power function x^y [Din+13].

Using a Global Number of Guard Bits Versus Using Divide-and-Conquer

For large precisions, the faithful computation of e^Y to precision $-w_F - g$ may require many more guard bits, hidden within the $[e^Y]$ operator, and removed in its final rounding to precision $-w_F - g$. There are also probably more guard bits hidden in the faithful constant multipliers.

In this presentation, we chose this divide-and-conquer approach that matches the “computing just right” philosophy. It should be noted, however, that most previous works present a flattened architecture, where a single integrated error analysis leads to a single, global number of guard bits (5 guard bits in [DD07], 8 in [PS09], for instance). The FPExp code of FloPoCo at the time of writing this book also follows this method: its error analysis counts the errors in ulps entailed by the e^Y operator, sums it to the errors defined above, and determines the value of g out of the sum (from 3 to 5 guard bits depending on the precision).

It is clear that both approaches work, but it is as yet unclear which leads to the best architecture. This question (which is not specific to the exponential operator) remains to be studied quantitatively.

22.4 Fixed-Point-In, Floating-Point Out Exponential

This variant is actually a simplification of Fig. 22.4 that starts with X_{fix} . The nominal size of X_{fix} is a $\text{sfix}(w_E, -w_F)$ format that will match the IEEEfloat output size. In this variant, X_{fix} is exact, so the corresponding g LSB guard bits are zero.

There is a minor difference with the previous architecture, as X_{fix} is already signed. It saves the negate box but requires signed constant multipliers by $1/\log(2)$ and $\log(2)$.

22.5 Fixed-Point Computation of e^Y

Let us now turn to the computation of e^Y , with $Y \in [-\frac{1}{2}, \frac{1}{2}]$ an $\text{sfix}(-1, -w_F - g)$ number with $g \geq 3$. The output will be an approximation to $e^Y \in$

$[0.65, 1.65]$; therefore, its MSB has bit position 0: it will be an $\text{ufix}(0, -w_F - g)$ format (see Fig. 22.4).

For very small precisions, in particular for the very small formats used for machine learning such as BFLOAT, DLFLOAT, and MSFP8-11 (see Sect. 2.5), the simplest option is to tabulate e^Y . For instance, for BFLOAT, we have $w_F = 7$, so with $g = 3$, we need a table of 2^{10} entries of 11 bits. This table is more than four times smaller than the $2^{12} \times 15$ -bit table of Fig. 22.3. Still, with the architectural overhead of Fig. 22.4, it is unclear if this solution is competitive with the direct tabulation method of Sect. 22.2.4, all the more as the latter provides a correctly rounded result.

For larger precisions, many techniques have been proposed to evaluate the fixed-point exponential in hardware. They are reviewed in Muller's book [Mul16]. In this book, we focus on two families of techniques: Sect. 22.5.1 presents in detail a family of short-latency techniques based on tabulation and Taylor polynomials, which makes a very good use of the resources available in modern FPGAs. Section 22.5.5 then describes a very generic family of iterative algorithms which have a longer latency but possibly use lower resources on ASIC.

22.5.1 Using Large Tables and Square Multipliers

This technique is based on a second range reduction, splitting Y as illustrated in Fig. 22.5:

$$Y = A + Z \quad (22.19)$$

where A consists of the k most significant bits of Y (bit positions -1 to $-k$) and Z consists of the remaining lower bits (bit positions $-k - 1$ to $-w_F + g$). Then, we have

$$e^Y = e^{A+Z} = e^A \cdot e^Z. \quad (22.20)$$

Here, e^A will be tabulated in a read-only table indexed by A . For e^Z , we will exploit that Z is small:

$$0 \leq Z < 2^{-k} \quad (22.21)$$

by evaluating it using a Taylor development to the required order:

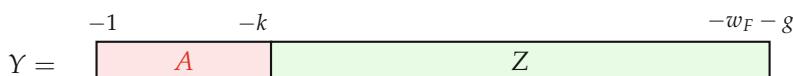


Fig. 22.5 Decomposition of Y into $A + Z$.

$$e^Z \approx 1 + Z + Z^2/2 + \dots \quad (22.22)$$

This formula has the advantage over the more advanced polynomial approximation techniques of Chap. 18 that the three first coefficients are powers of two; therefore, the corresponding multiplications will be constant shifts.

To exploit the Taylor formula, let us define

$$f(Z) = e^Z - 1 \quad (22.23)$$

$$h(Z) = e^Z - 1 - Z \quad . \quad (22.24)$$

Since $0 \leq Z < 2^{-k}$, Taylor entails

$$0 \leq f(Z) < 2^{-k+1} \quad (22.25)$$

$$0 \leq h(Z) < 2^{-2k} \quad . \quad (22.26)$$

In other words, the MSB position of $f(Z)$ is $-k$, and the MSB of $h(Z)$ has position $-2k - 1$.

From (22.25), we remark that the binary representation of $e^Z = 1 + f(Z)$ has at least $k - 1$ zeroes between the leading one and the next nonzero bit. A first idea is that we can save the multiplication by these $k - 1$ zeroes by implementing $e^A e^Z$ as

$$e^A \times e^Z = e^A + e^A \times f(Z) \quad . \quad (22.27)$$

There, the LSB position of the two addends should be $-w_F - g$.

A second idea is that $f(Z)$ can be evaluated as

$$f(Z) = Z + h(Z) \quad . \quad (22.28)$$

Here, $h(Z)$ captures all the expensive part of the computation of e^Z (all the multiplications of the Taylor series) in a term that can be $2k$ bit smaller than e^Z since $h(Z) < 2^{-2k}$.

Figure 22.6 shows an architecture that exploits this decomposition: it first evaluates $H \approx h(Z)$ in the $[e^Z - Z - 1]$ box (e.g., using any of the generic methods of Part III of this book), then builds $F \approx f(Z)$ by adding Z , and then implements the multiplication by $T \approx e^A$ as per (22.27).

This figure introduces several more parameters:

- $\ell = -w_F - g$ is used to lighten notations.
- The parameter g' is a number of guard bits used internally to ensure an accurate enough computation of e^Y . As discussed previously, an alternative is to use a single global guard bit parameter g . By developing the error analysis as a function of g and g' , we leave this choice open: the single- g option simply uses $g' = 0$.

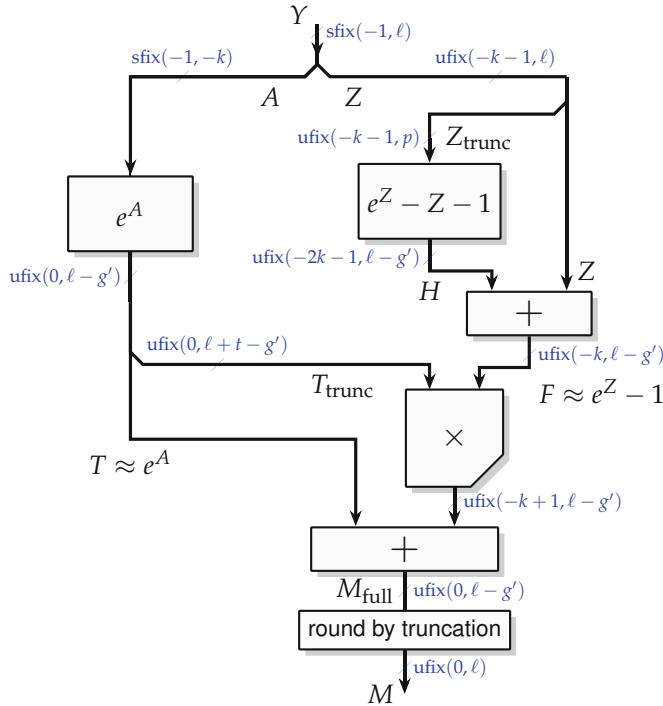


Fig. 22.6 Higher-order range reduction architecture.

- The parameter t captures the potential of truncation of e^A in the computation of $e^A \times f(Z)$. A truncated multiplier may be used there, and then its two inputs should be roughly the same size (see Sect. 8.1.3); therefore, it is possible to truncate e^A before the multiplier to the size of $F \approx f(Z)$, keeping only its most significant bits. This is the rough idea; the exact choice of the value of t will be deduced from the error analysis.
- The parameter p captures the potential of truncation of Z before the computation of $h(Z)$. Recall that the MSB of $H \approx h(Z)$ has position $-2k - 1$. Its LSB will have the position $\ell - g'$. The size of H is therefore $-2k - \ell + g'$. As a consequence, we do not need to compute it out of all the bits of Z : we may use, as input to the $[e^Z - Z - 1]$ box which evaluates $h(Z)$, a truncation Z_{trunc} of Z to its most significant bits. We define the parameter p as the LSB of Z_{trunc} . As for t , its exact value will be deduced from the error analysis.

Let us now develop the error analysis for the evaluation of e^Y by following the architecture of Fig. 22.6 from its output to its input:

$$\begin{aligned}
\delta_{\text{arch}} &= M - e^Y \\
&= M - e^A e^Z \quad \text{since the decomposition } Y = A + Z \text{ is exact} \\
&= M - M_{\text{full}} \\
&\quad + M_{\text{full}} - (T + T_{\text{trunc}}(Z + H)) \\
&\quad + (T + T_{\text{trunc}}(Z + H)) - (T + T(Z + H)) \\
&\quad + T(1 + (Z + H)) - T(1 + (Z + h(Z_{\text{trunc}}))) \\
&\quad + T(1 + Z + h(Z_{\text{trunc}})) - T(1 + Z + h(Z)) \\
&\quad + T(1 + Z + h(Z)) - Te^Z \\
&\quad + Te^Z - e^A e^Z \\
&= \delta_{\text{final round}} + \delta_{\text{mult}} + \delta_{\text{trunc}T} + \delta_H + \delta_{\text{trunc}Z} + 0 + \delta_T \quad (22.29)
\end{aligned}$$

where the error terms can be summarized as follows (defining $u = 2^{\ell-8'}$ the value of the ulp of the datapath): The error terms can be summarized as follows:

- $\delta_{\text{final round}} = M - M_{\text{full}}$ is the error of rounding off the g' guard bits. Its bound is $\bar{\delta}_{\text{final round}} = 0$ if $g' = 0$. Otherwise, it is a round to nearest, bounded by $2^{\ell-1}$. As usual, we implement this rounding by truncation for free, provided the rounding bit $2^{\ell-1}$ has been added to all the entries of the e^A table.
- $\delta_{\text{mult}} = M_{\text{full}} - (T + T(Z + H))$ is the rounding error due to the multiplier and is bounded by u for a truncated faithful multiplier.
- $\delta_{\text{trunc}T} = (T + T_{\text{trunc}}(Z + H)) - (T + T(Z + H)) = (T_{\text{trunc}} - T)(Z + H)$ is the error corresponding to the truncation of T (bounded by $2^{l+t-g'}$), amplified by $Z + H = F \approx e^Z - 1$ (roughly bounded by $1.25 \cdot 2^{-k}$ using Taylor formula). Its order of magnitude should be u so that the error sources are balanced: $2^{l+t-g'} \cdot 1.25 \cdot 2^{-k} \approx 2^{\ell-8'}$ suggests $t = k$ for $\bar{\delta}_{\text{trunc}T} = 1.25u$ or $t = k - 1$ for $\bar{\delta}_{\text{trunc}T} = 0.625u$.
- $\delta_H = T(1 + (Z + H)) - T(1 + (Z + h(Z_{\text{trunc}}))) = T(H - h(Z_{\text{trunc}}))$ is the overall error (amplified by T) due to the component evaluating $h(Z) = e^Z - Z - 1$. It includes the approximation error. If this function is tabulated, δ_H can be bounded by $\bar{\delta}_H = 1.65u/2$. This function can also be delegated to a faithful generic function approximator, in which case $\bar{\delta}_H = 1.65u$. Other architectural variants, using Taylor further, are also possible, each with its $\bar{\delta}_H$.

One more trick can be used here. Remember that the range reduction architecture needs a smaller $\bar{\delta}_{\text{arch}}$ when $e^Y < 1$. In this case, we also have $e^A \leq e^Y < 1$ (since $A \leq Y$ and e^x is monotonically increasing); therefore, we can use the bound $T < 1$ instead of $T < 1.65$.

- $\delta_{\text{trunc}Z} = T(h(Z_{\text{trunc}}) - h(Z))$ is the error (also amplified by T) due to truncating Z . With a truncation to LSB p , we have $Z = Z_{\text{trunc}} + e$ with $0 \leq e < 2^p$; then, Taylor on h gives $h(Z_{\text{trunc}}) - h(Z) \approx eh'(Z) = eZ + \dots$; hence, $|Z| < 2^{-k}$ entails the bound $\bar{\delta}_{\text{trunc}Z} = 2 \cdot 2^{p-k}$ where the factor 2

- accounts for the higher-order terms and the amplification by T . For instance, with the choice $p = \ell - g' + k - 1$, we have $\bar{\delta}_{\text{trunc}Z} = 2^{\ell-g'} = u$.
- $T(1 + Z + h(Z)) - Te^Z = 0$ since $h(Z)$ is exactly defined as $h(Z) = e^Z - Z - 1$. Again, the approximation error is accounted for in $\bar{\delta}_H$.
 - $\bar{\delta}_T = (T - e^A)e^Z$ is the error due to the tabulation of e^A . With rounding to the nearest, it can be bounded by $\bar{\delta}_T = u/2 \cdot 1.25$ as previously, where the 1.25 is a rough bound on e^Z .

Summing all these terms, we have $\bar{\delta}_{\text{arch}}$ between $3u$ and $6u$, depending on the architectural choices made, plus $\delta_{\text{final round}}$.

If we keep $g' = 0$, then the objective should be to keep $\bar{\delta}_{\text{arch}}$ just below $5u$, so that with the $3u$ of the floating-point range reduction, we may use $g = 4$ guard bits. For instance,

- For a small precision where we can tabulate a correctly rounded $h(Z)$, the choice of $p = \ell + k - 1$ and $t = k$ leads to $\bar{\delta}_{\text{arch}} = 0 + 1 + 1.25 + 1.65/2 + 1 + 0.625 = 4.7u$;
- For larger precisions where we have a faithful rounding of $h(Z)$, we must use a slightly more accurate truncation of T with $t = k - 1$, entailing $\bar{\delta}_{\text{arch}} = 0 + 1 + 0.625 + 1.65 + 1 + 0.625 = 4.9u$.

If we choose to keep g to its minimal value of 3, then we need $g' > 0$. It is desirable in this case to select an architectural variant such that $\bar{\delta}_{\text{mult}} + \bar{\delta}_{\text{trunc}T} + \bar{\delta}_H + \bar{\delta}_{\text{trunc}Z} + \bar{\delta}_T < 4u$; then, $g' = 3$ can be used.

The trade-off here is between more computations in the floating-point range reduction and more computations in the evaluation of e^Y .

22.5.2 First-Order Architecture

For small precisions, it is possible to approximate $f(Z)$ as $f(Z) \approx Z$ (again using Taylor). The approximation error in this case is equal to the sum of all the neglected terms of the Taylor approximation:

$$\delta_{\text{approx},1}(Z) = 1 + Z - e^Z = -Z^2/2 - Z^3/6 - \dots \quad (22.30)$$

Its absolute value is slightly larger than the second-order term $Z^2/2$, which verifies $0 \leq Z^2/2 < 2^{-2k-1}$. This proves the bound

$$\bar{\delta}_{\text{approx},1} = 2^{-2k-1} (1 + 2^{-k}) \quad (22.31)$$

where the factor $1 + 2^{-k}$ is a gross overapproximation of the bound of the sum of higher-order terms of the Taylor series. To fix ideas, this approximation makes sense when $\bar{\delta}_{\text{approx},1}$ is of the order of one ulp, or $2^{-2k-1} \approx 2^\ell$, or $k \approx (\ell + 1)/2$. Assuming we want to keep $g = 3$ guard bits in the floating-point range reduction, the constraint on k is $k \geq (w_F + 4)/2$.

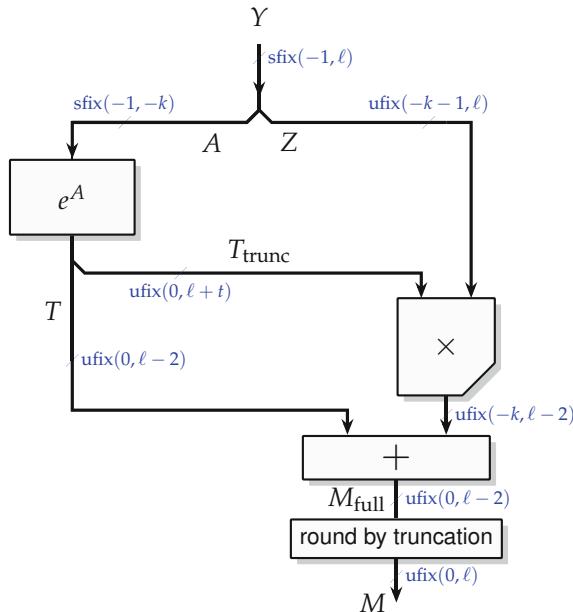


Fig. 22.7 A first-order architecture for e^Y that allows $g = 3$ in Fig. 22.4.

For instance, for binary16 floating point, we have $w_F = 10$, so the input Y is a $w_F + 3 = 13$ -bit number ($\ell = -13$), which may be split into A of $k = 7$ bits and Z of 6 bits. In this case, we may truncate e^A to LSB $\ell + k - 1 = -7$. The multiplier size in this case is 6×8 bits.

For IEEE single precision ($w_F = 23$), we would need $k = 14$ and a multiplier of 13×12 bits. We now have a fairly large table for e^A : it is probably better in this case to use the previous higher-order architecture.

The architecture is given in Fig. 22.7, and we leave it as an exercise to the reader, using a simplified version of Eq. (22.29), to show that it fulfills the constraint on $\bar{\delta}_{\text{arch}}$ that allows for $g = 3$.

22.5.3 Polynomial Approximation for Large Precisions

For larger values of w_F , the generic polynomial evaluator presented in Chap. 18 may be used as a black box to approximate $f(Z)$. It inputs a function of $[0, 1] \rightarrow [0, 1]$ (here, $h(x) = e^{2^{-k}x} - 2^{-k}x - 1$) with its input and output precisions (given on Fig. 22.6) and a degree and implements a faithful piecewise polynomial approximation. As this approach is also based on tables and multipliers, it is well suited to modern heterogeneous FPGAs.

We now have two parameters to set: k , which fixes the input to the e^A table, and the degree d of the polynomial, which fixes the trade-off between tables and multipliers in the polynomial evaluator. For illustration, for double precision, a sensible choice is $k = 9$ and $d = 2$, which needs 512 (2^9) intervals in the piecewise approximation. This also illustrates that this method scales quite well to large precisions.

The best trade-off is technology-dependent, and on FPGAs, it also depends on the resources consumed by the rest of the application.

The FloPoCo FPExp operator provides a good default choice of these parameters, and an expert mode allows the user to set them manually for different trade-offs.

22.5.4 FPGA-Specific Remarks

On FPGAs with embedded memories and DSP blocks, the former can be used to implement the e^A table, and the latter can be used to implement the multipliers of Figs. 22.6 and 22.7, along with the surrounding adders. Besides, these blocks should be used to their full:

- The value of t determined so far can be reduced for free, as long as T_{trunc} fits one DSP input. This reduces the error term $\delta_{\text{trunc}T}$.
- The output size of the e^A table can be enlarged for free to the maximum output size of the embedded memory (or memories) used.
- If the multiplier fits in a single DSP block, then this block computes the exact product which can then be rounded to the nearest. This reduces the error term δ_{mult} . The half-up addition needed for rounding to the nearest simply enlarges by 1 bit the final adder of Figs. 22.6 and 22.7.

The exact benefit of these remarks is dependent on the target precisions and the memory and DSP sizes.

An interesting case study is IEEE single precision ($w_F = 23$), choosing $g' = 0$ and $g = 4$ in the previous architecture, so $\ell = -27$. Then, the value $k = 10$ allows for a highly efficient architecture on most recent FPGAs with the choices $p = \ell + k - 1 = -18$ and $t = k = 10$ (made possible since $h(Z)$ will be tabulated with correct rounding):

- Both inputs to the multiplier are 18 bits, a size supported by all common DSP blocks. On FPGAs with asymmetric DSP blocks, e. g., the Xilinx 25 × 18 ones, the truncation of T_{trunc} can be relaxed to improve the accuracy.
- The size of the e^A table is $2^{10} \times 28$ bits, and the size of the $h(Z)$ table is $2^8 \times 6$ bits. Both tables can be grouped in a 36 Kbit block RAM used in $2^{10} \times 36$ dual-port mode.

22.5.5 Iterative Computation of e^Y Using Small Tables and Rectangular Multipliers

There is [Mul16] a large family of methods for computing exponential or logarithm that relies on preexisting (typically tabulated) pairs of reals $(E_i^\dagger, L_i^\dagger)$ such that $\forall i E_i^\dagger = e^{L_i^\dagger}$. Then, starting from seed values L_0 and E_0 such that $E_0 = e^{L_0}$, we may define two new real sequences (E_i) and (L_i) as follows:

$$\forall i > 0 \begin{cases} L_{i+1} = L_i + L_i^\dagger \\ E_{i+1} = E_i \times E_i^\dagger \end{cases} \quad (22.32)$$

The key point is that this iteration maintains the invariant $E_i = e^{L_i}$, since $E_0 = e^{L_0}$ and $E_{i+1} = E_i^\dagger E_i = e^{L_i^\dagger} e^{L_i} = e^{L_i^\dagger + L_i} = e^{L_{i+1}}$.

In this iteration, there is one addition and one multiplication. Since the former is cheaper in hardware than the latter, the preexisting pairs are chosen in such a way that the multiplication remains cheap, typically by using values of E_i^\dagger that fit on few bits.

This iteration can compute either logarithm or exponential. Let us first briefly sketch the case of the former. To evaluate the logarithm of a value E_0 , the idea is to ensure that $E_i \rightarrow 1$. For this purpose, each iteration selects, among the possible E_i^\dagger , an approximation to the inverse of E_i (again, this approximation should fit on few bits). Stopping at iteration n such that $E_n \approx 1$, we may initialize $L_n \approx 0$ and roll back (22.32) for decreasing i , that is, $L_i = L_{i+1} - L_i^\dagger$, until we have $L_0 \approx \log(E_0)$. Of course, as the addition is associative, it is also possible to initialize $L'_0 = 0$ and compute $L'_{i+1} = L'_i - L_i^\dagger$ for increasing i , so that $L'_n \approx \log(E_0)$.

To evaluate the exponential of a value L_0 , the iteration is similarly built such that $L_i \rightarrow 0$. Let us detail one simple version of this iteration that computes the exponential of our fixed-point operand $Y \in [-\frac{1}{2}, \frac{1}{2}]$. The seed value is $L_0 = Y$. Each iteration will start with a fixed-point number L_i and compute L_{i+1} closer to 0, until L_n is small enough that $e^{L_n} - 1$ may be accurately evaluated thanks to a simple table or a Taylor approximation.

To ensure that L_i converges to 0, we use the subword A_i consisting of the α_i leading bits of L_i (see Fig. 22.8) to address two tables. The first table holds $X_i^\dagger = E_i^\dagger - 1$, an approximation of $e^{L_i} - 1$ obtained by considering only the bits of A_i . Besides, this approximation is rounded to a fixed-point format of only α_i bits. The second table holds $L_i^\dagger = \log(E_i^\dagger)$, rounded to least significant bit (LSB) ℓ , where ℓ is a parameter to be determined by error analysis.

By construction, L_i^\dagger is quite close to L_i . One may check that computing

$$L_{i+1} = L_i - L_i^\dagger \quad (22.33)$$

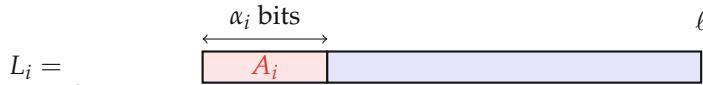


Fig. 22.8 Decomposition of L_i .

will result in cancelling the $\alpha_i - 1$ most significant bits of L_i . In such a sequence of additions, all the L_i and L_i^\dagger share the same LSB l . The number L_{i+1} fed into the next iteration therefore has $\alpha_i - 1$ fewer bits than L_i .

For the reconstruction of the exponential, we use (22.32) with decreasing i . However, a small change of variable is needed: Since $E_i \rightarrow 1$, we will save bits by storing $X^\dagger = E_i^\dagger - 1$ and computing $X_i = E_i - 1$. Then, the reconstruction matching (22.33) is (for decreasing i)

$$X_i = X_{i+1} \times X_i^\dagger + X_{i+1} + X_i^\dagger \quad (22.34)$$

and finally $e^{L_0} \approx X_0 + 1$. Here, X_{i+1} comes from the previous iterations, and X_i^\dagger is an α_i -bit number, so the product needs a rectangular multiplier for each iteration. The cumulated size of all these rectangular multipliers will be only slightly larger than that of one single full significand multiplier.

This architecture matches well to FPGAs that are only LUT-based. There, choosing α_i equal to the LUT input size for most iterations will optimize resource consumption. A larger α_i will entail larger tables but fewer iterations. However, this architecture is a poor match to modern FPGAs with their DSP and block memories. If the α_i are chosen to optimize the use of memories or LUTs, then the rectangular multipliers underuse the DSP blocks. If the α_i are chosen to match the DSPs, then the memory costs become very high. For this reason, on these modern FPGAs, the approaches in previous sections should be preferred.

For ASIC implementations, the delays of the additions on the L_i datapath (22.33) become a problem, since they are in the critical path. The solution is to use high-radix redundant arithmetic [Erc73; PEB03; PEB04], and we refer the interested reader to [PEB03; PEB04] for details. Unfortunately, a detailed comparison on the same technology of this iterative architecture and a polynomial-based one is currently missing.

22.5.6 To Read Further

Many more options for computing exponentials in hardware are presented in Muller's book [Mul16]. In particular, the classic CORDIC iteration computing the exponential and the logarithm can be considered variants of (22.32) where the convergence is achieved one bit at a time. Even the com-

putations of sines and cosines using CORDIC can be considered a variation on (22.32) computing the complex exponential.

High-performance single-precision implementations of exponential and logarithm can also exploit the floating-point capable DSP blocks on recent Intel FPGAs [LP17].

References

- [AS64] Milton Abramowitz and Irene A. Stegun. *Handbook of mathematical functions*. Applied Math. Series 55. National Bureau of Standards, Washington, D.C., 1964. (cit. on p. 642).
- [DD07] Jérémie Detrey and Florent de Dinechin. “Parameterized floating-point logarithm and exponential functions for FPGAs”. In: *Microprocessors and Microsystems, Special Issue on FPGA-based Reconfigurable Computing* 31.8 (2007), pp. 537–545. (cit. on p. 654).
- [Din+13] Florent de Dinechin, Pedro Echeverría, Marisa López-Vallejo, and Bogdan Pasca. “Floating-Point Exponentiation Units for Reconfigurable Computing”. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 6.1 (2013). (cit. on pp. 641, 654).
- [Ech+08] Pedro Echeverría, David Thomas, Marisa López-Vallejo, and Wayne Luk. “An FPGA Run-Time Parameterisable Log-Normal Random Number Generator”. In: *Reconfigurable Computing: Architectures, Tools and Applications*. Vol. 4943. Lecture Notes in Computer Science. Springer, 2008, pp. 221–232. (cit. on p. 641).
- [EL08] Pedro Echeverría and Marisa López-Vallejo. “An FPGA Implementation of the Powering Function with Single Precision Floating-Point Arithmetic”. In: *Real Numbers and Computers*. 2008. (cit. on p. 641).
- [Erc73] Miloš D. Ercegovac. “Radix-16 Evaluation of Certain Elementary Functions”. In: *IEEE Transactions on Computers* C-22.6 (1973), pp. 561–566. (cit. on p. 663).
- [Fra+17] Nicholas J. Fraser, Junkyu Lee, Duncan J. M. Moss, Julian Faraone, Stephen Tridgell, Craig T. Jin, and Philip H. W. Leong. “FPGA Implementations of Kernel Normalised Least Mean Squares Processors”. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 10.4 (2017). (cit. on p. 641).
- [HPS13] Matthew J. Hibbard, Eric R. Peskin, and Ferat Sahin. “FPGA implementation of particle swarm optimization for Bayesian network learning”. In: *Computers & Electrical Engineering* 39.8 (2013), pp. 2454–2468. (cit. on p. 641).

- [Jin+12] Qiwei Jin, Diwei Dong, H.T. Tse Anson, Gary C.T. Chow, David B. Thomas, Wayne Luk, and Stephen Weston. "Multi-Level Customisation Framework for Curve Based Monte Carlo Financial Simulations". In: *International Symposium on Applied Reconfigurable Computing*. Springer. 2012, pp. 187–201. (cit. on p. 641).
- [KD09] Nachiket Kapre and Andre DeHon. "Accelerating SPICE Model-Evaluation using FPGAs". In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2009, pp. 37–44. (cit. on p. 641).
- [Le+16] Julien Le Maire, Nicolas Brunie, Florent de Dinechin, and Jean-Michel Muller. "Computing floating-point logarithms with fixed-point operations". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2016. (cit. on p. 643).
- [Li+01] R.-C. Li, P. Markstein, J. P. Okada, and J. W. Thomas. *The Libm library and floating-point arithmetic for HP-UX on Itanium*. Tech. rep. Hewlett-Packard company, 2001. (cit. on p. 646).
- [LP17] Martin Langhammer and Bogdan Pasca. "Single Precision Logarithm and Exponential Architectures for Hard Floating-Point Enabled FPGAs". In: *IEEE Transactions on Computers* 66.12 (2017), pp. 2031–2043. (cit. on p. 664).
- [Mar00] Peter Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000. (cit. on p. 646).
- [Mik+18] Mantas Mikaitis, Dave Lester, Delong Shang, Steve Furber, Gengting Liu, Jim Garside, Stefan Scholze, Sebastian Höppner, and Andreas Dixius. "Approximate Fixed-Point Elementary Function Accelerator for the SpiNNaker-2 Neuromorphic Chip". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2018. (cit. on p. 641).
- [MMN13] Juan Carlos Moctezuma, Joseph P. McGeehan, and Jose Luis Nunez-Yanez. "Numerically Efficient and Biophysically Accurate Neuroprocessing Platform". In: *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. IEEE, 2013. (cit. on p. 641).
- [Mul+18] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiod, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. 2nd ed. Birkhäuser Boston, 2018. (cit. on p. 648).
- [Mul16] Jean-Michel Muller. *Elementary Functions, Algorithms and Implementation*. 3rd ed. Birkhäuser Boston, 2016. (cit. on pp. 655, 662, 663).
- [OS05] Stuart F. Oberman and Ming Y. Siu. "A High-Performance Area-Efficient Multifunction Interpolator". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2005. (cit. on p. 641).

- [PEB03] José-Alejandro Piñeiro, Miloš D. Ercegovac, and Javier D. Bruguera. “On-Line High-Radix Exponential with Selection by Rounding”. In: *International Symposium on Circuits and Systems (ISCAS)*. 2003. (cit. on p. [663](#)).
- [PEB04] J. A. Piñeiro, Miloš D. Ercegovac, and Javier D. Bruguera. “Algorithm and Architecture for Logarithm, Exponential, and Powering Computation”. In: *IEEE Transactions on Computers* 53.9 (2004), pp. 1085–1096. (cit. on p. [663](#)).
- [PS09] Robin Pottathuparambil and Ron Sass. “A parallel/vectorized double-precision exponential core to accelerate computational science applications”. In: *Field Programmable Gate Arrays*. ACM, 2009, pp. 285–285. (cit. on p. [654](#)).
- [Tan89] Ping Tak Peter Tang. “Table-Driven Implementation of the Exponential Function in IEEE Floating-Point Arithmetic”. In: *ACM Transactions on Mathematical Software* 15.2 (1989), pp. 144–157. (cit. on p. [646](#)).

Part V

Application Domains



CHAPTER 23

Arithmetic in the Design of Linear Time-Invariant Filters

This chapter provides an arithmetic-centered point of view on the construction of hardware digital filters. Such filters combine arithmetic components (adders and multipliers) with memory. The design of the arithmetic must take into account this memory. This chapter introduces the relevant concepts and then provides two definitions of faithfulness for hardware digital filters: faithful to an ideal linear time-invariant filter and faithful to a frequency specification. With these definitions, filter design and implementation become a single global optimization problem.

Digital signal processing (DSP) is a domain in itself. When implemented in hardware, DSP makes extensive use of the application-specific components introduced in this book. For instance, the bipartite table method (see Sect. 17.2, p. 503) was invented in the DSP community in the specific case of the sine function [Sun+84] a decade before it was rediscovered independently in the computer arithmetic community for the reciprocal function [DM95]. Similarly, much of the content of this book is relevant to hardware DSP practitioners.

This chapter focuses on digital filters, where the “computing just right” philosophy can be translated to specific methodologies that do not reduce to the ones already introduced.

Digital filters are essential components of modern technology, from medical equipment and scientific instruments to radar and navigation systems, from Hi-Fi audio and radio-enabled personal electronics to Internet of Things devices. Filter design is therefore a core topic in digital signal processing (DSP) and control theory, one that has received significant research interest for half a century. Hardware digital filters are found in many application-specific integrated circuits (ASICs), and some application domains (5G/6G backbones, autonomous vehicles, edge computing) rely on hardware filters implemented on field programmable gate arrays (FPGAs).

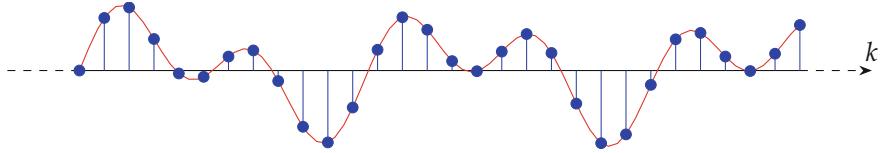


Fig. 23.1 A discrete-time signal (in blue) sampling a continuous signal (in red).

23.1 An Introduction to Discrete-Time Signals and Filters

This section is a very short introduction to the signal processing notions that will be useful in this chapter. Readers with a DSP background may skip it, or just browse it for the notations used in later sections. For other readers, it should make the chapter self-sufficient, but it is by no means an extensive introduction to the topics treated here. DSP textbooks [[Ant05](#); [Smi22](#); [OS09](#); [VKG14](#)] will provide the missing proofs and mathematical derivations, and many more subjects not covered here.

23.1.1 Discrete-Time Signals

A discrete-time signal x is a sequence of real or complex data:

$$x = \{x(k)\}_{k \in \mathbb{Z}} \quad \text{where } x(k) \in \mathbb{R} \text{ or } x(k) \in \mathbb{C} . \quad (23.1)$$

In application domains such as audio or radio processing, the index k can be interpreted as the discrete time, and a signal is typically interpreted as the result of sampling a continuous value at regular intervals (see Fig. 23.1).

In other application domains, signals may have more than one discrete dimension. For instance, some image processing techniques consider an image as a two-dimensional signal $p(i, j)$ where the discrete indices i and j are the two spatial dimensions of the image (the pixel coordinates). A video flux combines one discrete time dimension and two discrete spatial dimensions. This chapter focuses on arithmetic issues, and from this point of view, the dimensionality of the signal support is mostly irrelevant. Therefore, for simplicity, the following focuses on one-dimension discrete-time signals.

23.1.2 Elementary Operations on Discrete-Time Signals

Two signals x and y can be added, and this simply corresponds to element-wise addition:

$$x + y = \{x(k) + y(k)\}_{k \in \mathbb{Z}} . \quad (23.2)$$

A signal x can be scaled by a real or complex constant c , and again this is an element-wise operation:

$$cx = \{cx(k)\}_{k \in \mathbb{Z}} . \quad (23.3)$$

Finally, a signal x can be time-shifted by a constant integer n , denoted as:

$$\mathcal{D}^n(x) = \{x(k - n)\}_{k \in \mathbb{Z}} . \quad (23.4)$$

If n is positive, one may describe $\mathcal{D}^n(x)$ as x delayed by n discrete time units (or samples). A time-shift by one (i.e., $n = 1$) is denoted as $\mathcal{D}(x)$.

These three basic operations are the foundation on which more complex signal operations will be built in the following.

23.1.3 Discrete-Time Systems or Filters

A discrete-time system, or discrete-time filter, is a mathematical object that inputs signals and output signals.

We focus in this chapter on filters that input only one signal u and output only one signal y (Fig. 23.2). This simplifying assumption is motivated by our focus on computer arithmetic. The generalization to multiple input, multiple output (MIMO) filters is not expected to always be straightforward.

As a mathematical object, the ideal filter inputs the whole sequence (Fig. 23.2, left), which is potentially infinite. However, a realization of a filter will typically input the signal one value at a time (Fig. 23.2, right). The interface then looks similar to that of numerical functions of Part III: just like the component of Fig. 16.2, p. 480, the component of Fig. 23.2 inputs one value, and outputs one value at a time. The essential difference is that the filter has some memory of the values it has been processed in the past, whereas the function's output only depends on its current input.

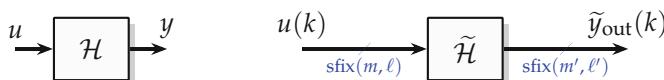


Fig. 23.2 An ideal filter \mathcal{H} (left) and its implementation $\tilde{\mathcal{H}}$ (right).

23.1.4 Linear Time-Invariant Filters

Among digital filters, a widely useful class is that of linear time-invariant (LTI) filters [Ant05].

A filter \mathcal{H} is said to be linear if for any two signals x and y and any real constant c ,

$$\mathcal{H}(x + cy) = \mathcal{H}(x) + c\mathcal{H}(y) \quad (23.5)$$

(this signal equality is an element-wise equality).

\mathcal{H} is said to be time-invariant iff for any signal x and any $n \in \mathbb{Z}$,

$$\mathcal{H}(\mathcal{D}^n(x)) = \mathcal{D}^n(\mathcal{H}(x)) , \quad (23.6)$$

i.e., the same signal is obtained by delaying the signal input to \mathcal{H} , or by delaying its output signal.

Time-invariance is an important property from an application point of view: when k is the discrete time, it means that the behavior of the filter does not depend on the moment at which it is used. This is indeed a desirable property to expect from a hardware component. For multidimensional signals, time-invariance can be generalized to *shift invariance* [VKG14]. For instance, a shift-invariant filter in image or video processing will have a behavior independent of the pixel position. Again, this chapter focuses on time-invariance for simplicity without loss of generality.

Linearity and time-invariance enable many powerful analysis and synthesis techniques, some of which are reviewed in this chapter.

23.1.4.1 Impulse Response of an LTI Filter

The simplest nonconstant signal is the *unit impulse* signal¹ shown in Fig. 23.3, and defined by

$$\delta(k) = \begin{cases} 1 & \text{for } k = 0 \\ 0 & \text{otherwise} \end{cases} . \quad (23.7)$$

Definition 23.1 The impulse response h of an arbitrary filter \mathcal{H} is defined as the output of \mathcal{H} when the input is the impulse δ :

$$h = \mathcal{H}(\delta) . \quad (23.8)$$

¹ It is also called (discrete) Dirac delta function as it has similar properties to its continuous counterpart. It is also a special case of Kronecker delta function and therefore sometimes named so.

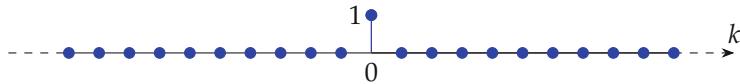


Fig. 23.3 The unit impulse signal.

An arbitrary signal u can be trivially rewritten as an infinite sum of scaled, time-shifted impulses:

$$\forall k \in \mathbb{Z} \quad u = \sum_{i \in \mathbb{Z}} u(i) \delta(k - i) \quad (23.9)$$

or

$$u = \sum_{i \in \mathbb{Z}} u(i) \mathcal{D}^i(\delta) \quad . \quad (23.10)$$

By linearity and time-invariance, the response of an LTI filter \mathcal{H} to an input signal u is a sum of scaled, time-shifted impulse responses:

$$y = \mathcal{H}(u) = \sum_{i \in \mathbb{Z}} u(i) \mathcal{D}^i(\mathcal{H}(\delta)) = \sum_{i \in \mathbb{Z}} u(i) \mathcal{D}^i(h) \quad . \quad (23.11)$$

The value of the output signal at discrete time k can thus be defined as

$$y(k) = \sum_{i \in \mathbb{Z}} u(i) h(k - i) \quad . \quad (23.12)$$

In other words, the behavior of an LTI filter \mathcal{H} on an arbitrary input signal u is completely defined by u and the impulse response of the filter, $h = \mathcal{H}(\delta)$, which is itself not a filter, but a signal.

In the usual case when the index k denotes the discrete time, it is common to impose the restriction that the filter should be *causal*, i.e., it does not react before it receives the impulse. In mathematical terms, this is simply written $\forall k < 0, h(k) = 0$.

23.1.4.2 Finite Impulse Response and Infinite Impulse Response Filters

The fact that an LTI filter is perfectly characterized by its impulse response leads to a first classification of LTI filters:

- If the impulse response takes a *finite* number of nonzero values, the filter is called a finite impulse response (FIR) filter.
- If the impulse response takes an *infinite* number of nonzero values, the filter is called an infinite impulse response (IIR) filter.

In other words, when input an impulse, an FIR filter will produce nonzero outputs for some time, then return to zero. Conversely, the output of some IIR filters will never return to a constant 0.

Note that the terms FIR and IIR only describe causal LTI filters.

23.1.4.3 Convolutions and Filter Structures

Equation (23.12) describes a *convolution*: the output of an LTI filter is the convolution of the input signal with the impulse response of the filter. This is typically denoted as $y = u \circledast h$. The convolution operator \circledast has many interesting properties:

- It is commutative: $a \circledast b = b \circledast a$. The proof is the change of variable $j = k - i$ in (23.12).
- It is distributive over addition: $a \circledast (b + c) = a \circledast b + a \circledast c$. The proof is again simple from (23.12).
- It is associative: $a \circledast (b \circledast c) = (a \circledast b) \circledast c$, which can therefore be denoted as $a \circledast b \circledast c$. The proof consists in reorganizing the double summation that appears when invoking (23.12) twice.

These proofs are only valid if all the infinite summations converge [VKG14].

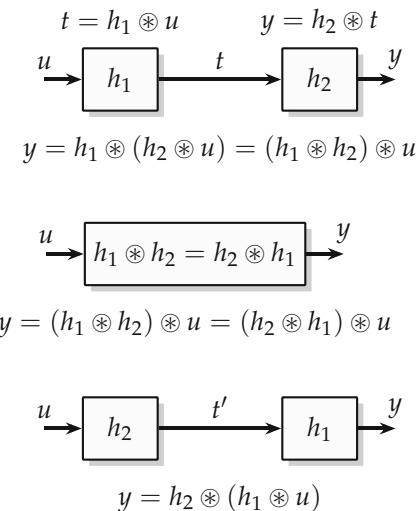


Fig. 23.4 Filter cascades: the three represented filters are mathematically equivalent.

Together with the graphical representation of filters, the convolution properties translate to powerful tools for composing them, as illustrated in Figs. 23.4 and 23.5. Remark that on these figures, each LTI filter is identified by its impulse response.

Several LTI filters in sequence are called a *filter cascade*. As Fig. 23.4 shows, cascades are associative and commutative, because each filter is a convolu-

tion with an impulse response, and these convolutions are associative and commutative. An important application will be seen in Sect. 23.1.5.

The distributivity of convolution over addition translates to the possibility to decompose a filter into a sum of two filters. This is illustrated by Fig. 23.5, which also explains why this decomposition is sometimes called *parallel combination*.

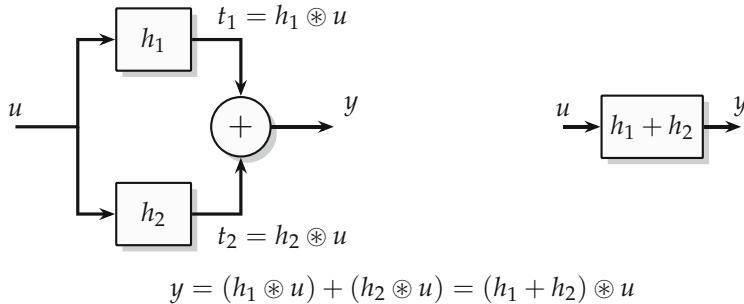


Fig. 23.5 Filter addition (also known as parallel decomposition): the two represented filters are mathematically equivalent.

It is very important to remind the reader that these properties are deeply rooted in the linearity property. In particular, actual addition or multiplication hardware involves some rounding, which is not linear. Therefore, the equivalent filters in Figs. 23.4 and 23.5 may give widely different results when implemented with finite-precision operators. Worse, a converging infinite summation may no longer converge if one adds to it an infinite number of rounding errors. This is the root cause of an issue called *limit cycle oscillations*, where a circuit supposedly implementing a stable filter exhibits a periodic oscillation instead of decaying to zero for a zero input after applying a specific stimulus. The main point of the present chapter is to frame this kind of problems and then to provide the tools that will help design filters that, in spite of this loss of linearity, behave like the ideal ones.

23.1.4.4 Stability

An important property that is desirable in filters is bounded-input, bounded-output (BIBO) stability (or stability for short).

Definition 23.2 A filter is BIBO-stable iff, for any bounded input signal u (meaning that $\exists \bar{u} \in \mathbb{R}$ such that $\forall k \in \mathbb{Z}, |u(k)| < \bar{u}$), the output signal is also bounded (usually with a different bound \bar{y}).

Tools that quantify BIBO stability (i.e., that provide the value of \bar{y} knowing \bar{u}) will be introduced in Sect. 23.3.1 when describing the error analysis of an LTI filter.

It is easy to see that FIR filters are always stable when their impulse response $h(k)$ in (23.12) has a finite number of nonzero and finite values. It can be shown that IIR filters are BIBO-stable iff their impulse response is absolutely summable, i.e., $\sum_{k \in \mathbb{Z}} |h(k)|$ converges.

Again, beware that the filters considered so far are ideal mathematical objects. The BIBO stability of a hardware digital filter implementing an ideal filter is one of the issues addressed in this chapter. Before that, it is important to describe how such hardware is designed.

23.1.5 Specifying LTI Filters by Constant-Coefficient Equations

It is convenient to define the behavior of a filter by a finite equation that relates output signal values and input signal values using only addition, multiplication and time shift. An example of such equation is

$$y(k) = \sum_{i=0}^n b_i u(k-i) - \sum_{i=1}^m a_i y(k-i). \quad (23.13)$$

Such an equation is sometimes called *difference equation* due to its similarity to the differential equations describing continuous filters.

An advantage of such an equation is that it can be straightforwardly translated into a *filter structure* (sometimes also named *signal-flow graph* or *data-flow diagram*). An example of filter structure is shown in Fig. 23.7, and many more examples will appear in the following pages, for instance, the filter structure corresponding to (23.13) is Fig. 23.9a, p. 679.

Filter structures only involve (in finite number) the three basic components presented in Fig. 23.6. These three components correspond to the three elementary signal transformations described by Eqs. (23.2), (23.3), and (23.4), respectively.

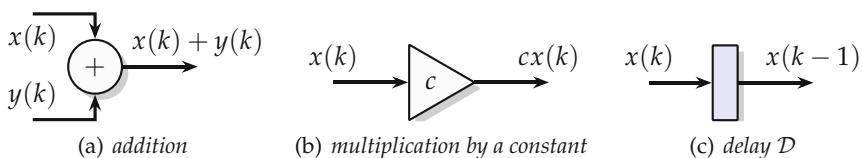


Fig. 23.6 The three abstract components used in filter structures. Note that each of these components is itself an elementary LTI filter.

A filter structure can then be translated to a hardware implementation by replacing the three components of Fig. 23.6, respectively, by adders, multipliers, and synchronous registers (i.e., registers which are all driven by a common clock). The main issue addressed in the sequel is the determination of the actual data formats used as inputs and outputs of these components. Indeed, filter structures are abstract architectures, in the sense that the shown wires carry real numbers, not machine numbers.

The direct interpretation of a difference equation as a signal-flow graph is sometimes called *parallel implementation*, because there are as many multipliers working in parallel as there are multiplications in the difference equation. A parallel implementation computes one output sample each clock cycle. It is also possible to build implementations that require several clock cycle to compute one sample (by time-sharing the arithmetic operators or using bit-serial arithmetic), or conversely implementations that compute several samples per clock cycle. The sequel focuses for simplicity on straightforward parallel implementations.

23.1.5.1 Non-recursive Filters

A non-recursive filter evaluates the sum in (23.12) for a finite number of elements and therefore directly provides the equation:

$$y(k) = \sum_{i=0}^n b_i u(k-i) \quad \text{where } b_i = h(i) \text{ and } n = \text{argmax}\{h(k) \neq 0\}. \quad (23.14)$$

Here, n is called the *order* of a filter. Note that the number of coefficients is $n+1$, which is sometimes called the *length* of a filter. Due to the finite length, non-recursive filters always belong to the class of FIR filters.

The filter structure if this equation is shown in Fig. 23.7 for $n = 4$. Its corresponding parallel implementation consists of a register chain and an sum of products by constants (SOPC).

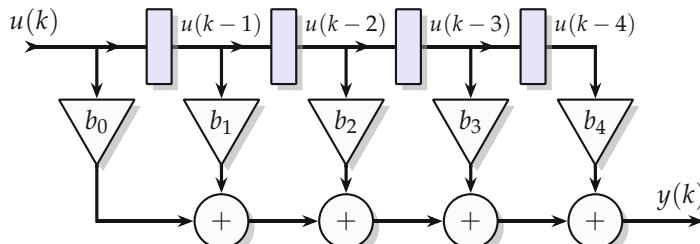


Fig. 23.7 Direct form non-recursive filter.

A surprising property of single input, single output (SISO) filter structures is the following: any SISO structure such as Fig. 23.7 can be *transposed* into an equivalent filter structure (see Sect. 12.5.3, p. 407): all the wire directions are reversed, branches are replaced with adders, and adders are replaced with branches. Applying this transposition to Fig. 23.7 leads to another non-recursive filter structure shown in Fig. 23.8. Its parallel implementation involves a multiple constant multiplication (MCM) followed by a chain of delays and adders. These adders are often called *structural adders* to distinguish them from the adders used to implement the MCM as per Chap. 12.

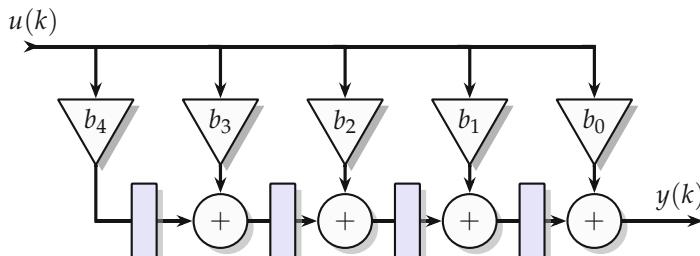


Fig. 23.8 Transposed non-recursive filter: it is obtained from Fig. 23.7 by exchanging the directions of all the wires, and replacing wire forks by adders and adders by wire forks.

23.1.5.2 Recursive Filters

Recursive filters are defined by the recursive difference equation

$$y(k) = \sum_{i=0}^n b_i u(k-i) - \sum_{i=1}^m a_i y(k-i), \quad (23.15)$$

where the coefficients a_i and b_i are constant (independent of k) so that this equation describes a time-invariant filter. Note that the form (23.15) ensures that the filter described by this equation is linear. Due to its recursive nature, applying (23.15) to an impulse signal typically leads to an output signal of infinite length: $y(k)$ will be computed from previous outputs as well, so as long as these do not cancel each other perfectly,² this equation typically describes an IIR filter.

The filter structure corresponding to (23.15) is called *direct form I* and is shown in Fig. 23.9a. By transposing the direct form I we will get the *transposed direct form I* as given in Fig. 23.9b. While the parallel implementation of

² There are very specific cases where this cancellation happens, the interested reader may have a look at Cascaded-Integrator-Comb (CIC) filters.

a direct form I contains an SOPC, the building block of the transposed direct form I is an MCM.

In addition, another form can be obtained by considering Fig. 23.9a as a cascade of two LTI filters and applying the commutativity of their underlying convolutions as per Fig. 23.4. One obtains an equivalent filter structure called the *direct form II*, illustrated by Fig. 23.10. This form is interesting because the delay results can be shared (it is easy to see in Fig. 23.10 that they contain the same values). In a parallel implementation, the delays are realized by registers, so half of them can be saved when $m = n$. Figure 23.11a shows a rearrangement of Fig. 23.10 that illustrates this sharing. This figure also shows that the parallel implementation of the direct form II consists of two independent SOPC. The parallel implementation of its transposed version, shown in Fig. 23.11b, similarly shares registers, and it involves two independent MCM.

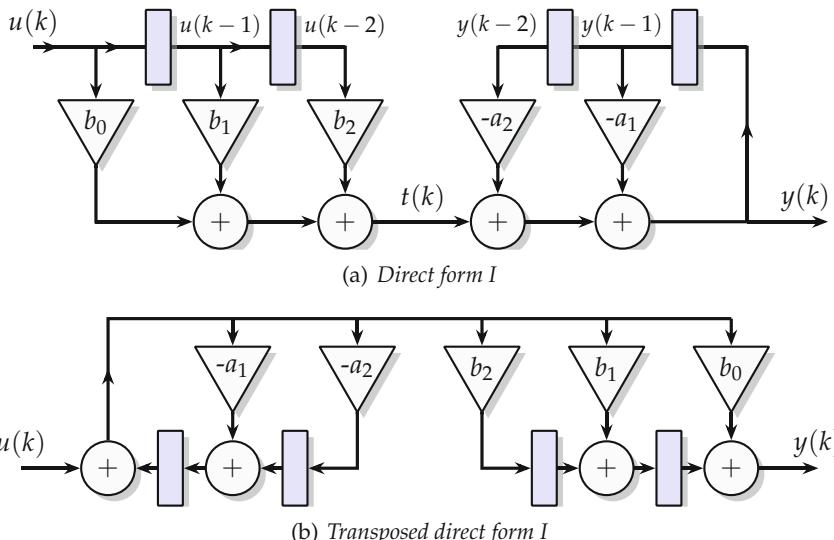


Fig. 23.9 Some recursive filter structures.

Many more filter structures exist. Some of them are built using frequency-domain considerations introduced in the sequel. However, the purpose of this chapter is not to cover all of them exhaustively [OS09, ch. 6], [Ant18, ch. 9]. Our focus will be on capturing and controlling the effect of finite precision so that *implementations* of these filters exhibit the behavior of the abstract filter structures in spite of the nonlinearities introduced.

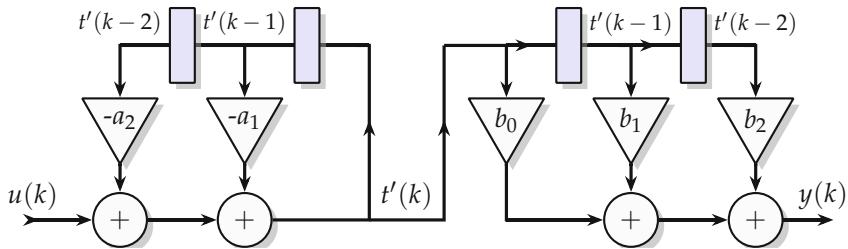
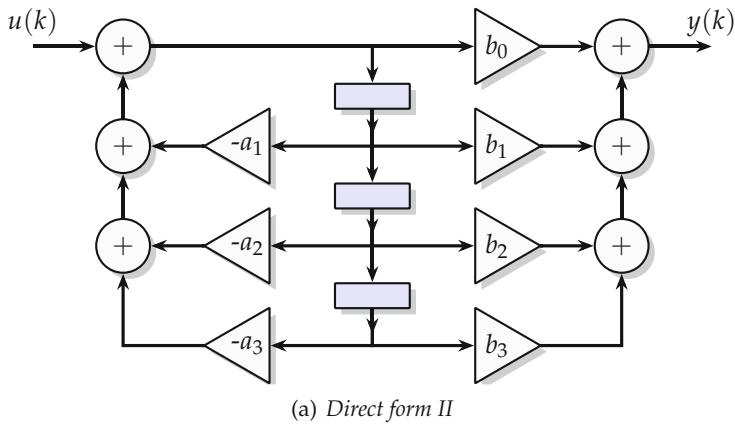
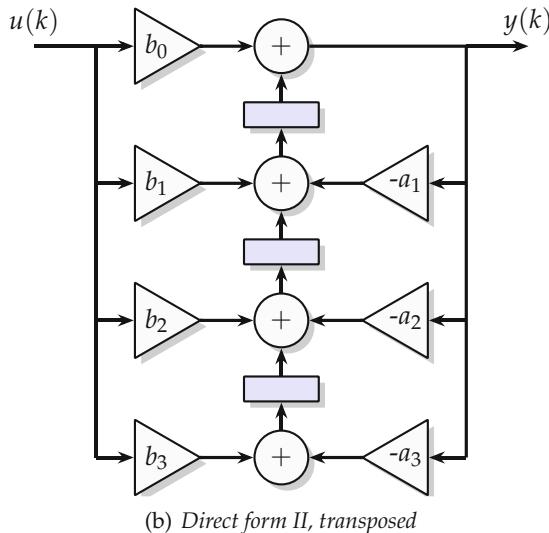


Fig. 23.10 Direct-form II filter structure.



(a) Direct form II



(b) Direct form II, transposed

Fig. 23.11 The direct form II with sharing of the delay elements.

23.1.6 Abstract Filter Structures Versus Finite-Precision Circuits

To motivate this focus, consider Figs. 23.9 and 23.11. First remark that each of these four architectures consists of a recursive loop involving the coefficients a_i , and a non-recursive forward path involving the coefficients b_i . The forward part can be managed with error analysis techniques already used in other parts of this book. A new difficulty is the recursion. Any rounding error happening there is fed back to the computation, where it will be combined with the other rounding errors in an endless loop, in an infinite accumulation of rounding errors. This issue will be captured more formally, and Sect. 23.3 will show how to make sure that this infinite error sum converges, and then that its value is small enough to be acceptable. This will, however, come at a cost. As in other chapters of this book, this will require the internal computations to use extended precision compared to the inputs/outputs.

To appreciate the impact of this, consider, for instance, the parallel implementation of the two equivalent forms of Figs. 23.9a and 23.10. A closer look shows that the multipliers by b_i are cheaper in the direct form I, since they input $u(k)$ there, while they input $t'(k)$ in the direct form II. Since $t'(k)$ is computed by the feedback loop, it may require many more bits of precision than $u(k)$ to absorb the infinite accumulation of rounding errors. A multiplier by b_i inputting $t'(k)$ is therefore typically larger than a multiplier by b_i inputting $u(k)$. Therefore, although it may seem at first sight that the direct form II of Fig. 23.11a is cheaper than the direct form I of Fig. 23.9a, thanks to the shared registers, a *stable implementation* using the direct form II may well turn out to be cheaper, thanks to its smaller multipliers. Note that the transposed direct form II of Fig. 23.11b multiplies the b_i by $u(k)$ as the direct form I, while sharing registers, so it seems to be the best combination. However, its critical path is longer (it goes through two multipliers and two adders), so it has its own drawback. To cut this critical path, it is possible to add one register to the wire labeled $t'(k)$ on Fig. 23.10, but then it is no longer possible to share registers.

All this hand-waving does not tell our reader which solution is best, and indeed the answer to this question will be very application-specific. Sections 23.2, 23.3, and 23.4 will show how to assess it quantitatively. They will, for instance, provide tools that, given the coefficients a_i and b_i , compute the minimum size of intermediate data (here $t'(k)$) that ensures the stability of the implementation.

23.1.7 Filters Directly Defined by Their Coefficients

There exist applications in which a filter is directly defined by a difference equation. As a first example, pulse-shaping filters (such as half-sine or root-raised cosine) are small FIR filters commonly used in wireless communica-

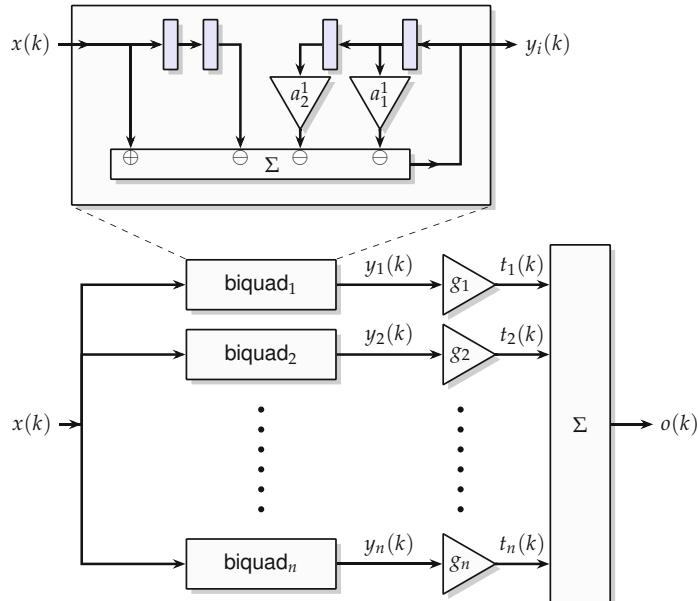


Fig. 23.12 Filter structure for multimodal additive sound synthesis.

cation [802.15.4]. Their coefficients are given by mathematical formulas depending only on the filter order.

Another example is physics-based multimodal additive sound synthesis, using the filter structure of Fig. 23.12. Here, a complex sound (e.g., the sound of a bell³) is decomposed into a weighted sum of decaying harmonics. Each harmonic can be defined by two parameters, its frequency f_i and its 60dB decay time T_i^{60} . These parameters and the gains g_i can be computed out of a physical model. Then each harmonic can be implemented by a simple second-order IIR whose coefficients a_i and b_i are defined by a simple function [Smi22, E.7] of f_i and T_i^{60} .

In most cases, however, the coefficients are computed out of a *frequency specification* in a process called *filter design* which we overview now.

23.1.8 Filters Defined in the Frequency Domain

The formulations (23.14) and (23.15) are *time-domain* descriptions, but an LTI filter is often specified by its frequency response (Fig. 23.13): an application requires a filter to pass or amplify signals in certain frequency ranges and attenuate them in other ranges.

³ This example and others can be heard on <https://faustide.grame.fr/>

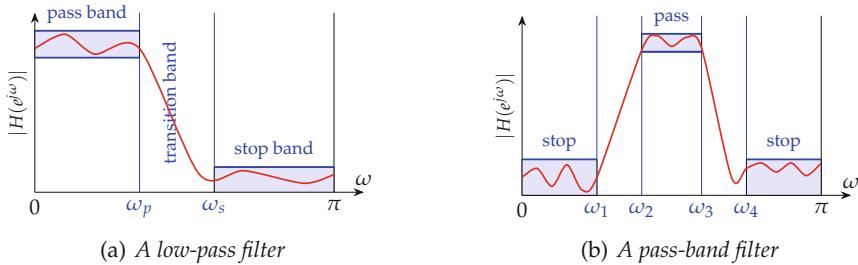


Fig. 23.13 Constraining the frequency response.

The general tool for relating the time domain and the frequency domain is the Fourier transform [OS09; VKG14]. Its discrete-time generalization is the z -transform \mathcal{Z} . It maps a signal x to a complex function $X = \mathcal{Z}(x)$, defined as

$$X(z) = \sum_{k=-\infty}^{+\infty} x(k)z^{-k} \quad \forall z \in \mathbb{C}. \quad (23.16)$$

For causal LTI filters the summation only considers positive k . In other words, $\mathcal{Z}(x)$ is a kind of (possibly infinite) polynomial in z^{-1} whose coefficients are the values of $x(k)$. One nice property of the z -transform is that when $X(z)$ is evaluated on the unit circle, i.e., for $z = e^{j\omega}$, it delivers the result of the discrete-time Fourier transform of the signal x . Here, $\omega = 2\pi f/f_s \in [0, 2\pi]$ is the radian frequency, which is the frequency normalized to the sampling frequency f_s and expressed in radian. Hence, $X(z)$ can be used as a frequency-domain description of the signal x .

It is easy⁴ to see that the z -transform is linear, and that a time shift of n samples corresponds to a factor of z^{-n} : $\mathcal{Z}(\mathcal{D}^n(x)) = \mathcal{Z}(x) \times z^{-n}$. For this reason, the delay of Fig. 23.6, whose role is to (time-)shift a signal by one sample, is often illustrated with a box $[z^{-1}]$ in the literature. Another useful property of the z -transform is that the convolution in the time domain corresponds to a product in the frequency domain. This is illustrated by Fig. 23.14. The z -transform $H(z)$ of the impulse response h of a filter is called the *transfer function* of the filter.

This point of view on filters is extremely productive for several reasons.

Firstly, thanks to the relationship with the Fourier transform, the frequency response of the filter is defined by the behavior of H for $z = e^{j\omega}$. For instance, a desired frequency response can be specified as a set of frequency bands, as illustrated in Fig. 23.13. Furthermore, the fact that $H(e^{j\omega})$ is a complex number captures the effect of a filter on both phase and amplitude: the amplitude response is the magnitude of $H(e^{j\omega})$, and the phase response is its polar angle. For instance, filters with a linear phase response

⁴ ... if you forget that all these are infinite sums and that their convergence is not obvious, see textbooks [OS09; VKG14] for a detailed analysis.

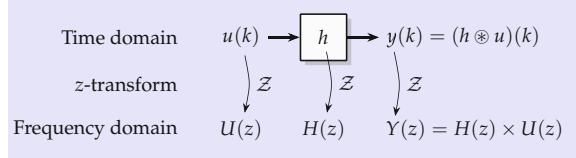


Fig. 23.14 Time domain, frequency domain and z -transform.

can be associated with FIR filters presenting some symmetry on their coefficients [OS09; VKG14].

Secondly, the associativity of time-domain convolution corresponds (by z -transform) to the associativity of the complex multiplication. Therefore, Fig. 23.14 also hints that a filter cascade can be obtained by decomposing a transfer function $H(z)$ into some product of two other functions. For instance, the transfer function of a causal FIR filter is a polynomial in z^{-1} (since (23.16) is finite), and the various ways to factor this polynomial provide many filter cascades [MW15].

For IIR filters, the z -transform of (23.15) leads to

$$Y(z) = \sum_{i=0}^m b_i U(z^{-i}) - \sum_{i=1}^n a_i Y(z^{-i}) \quad (23.17)$$

$$= \sum_{i=0}^m b_i z^{-i} U(z) - \sum_{i=1}^n a_i z^{-i} Y(z) \quad (23.18)$$

which can be identified to $Y(z) = H(z) \times U(z)$ (from (23.14)) to obtain:

$$H(z) = \frac{\sum_{i=0}^m b_i z^{-i}}{1 + \sum_{i=1}^n a_i z^{-i}}, \quad z \in \mathbb{C}, \quad a_i, b_i \in \mathbb{R}. \quad (23.19)$$

This is a rational transfer function. The stability of the filter can be related to the position of the poles of $H(z)$, i.e., the values of z for which the denominator $1 + \sum_{i=1}^n a_i z^{-i}$ is zero. An IIR filter is BIBO-stable iff its poles are within the unit circle⁵. Many interesting IIR filters have their poles close to the unit circle, which corresponds to a long-term memory effect in the feed-

⁵ A complete proof is out of the scope of this chapter, but we may give an intuition. $H(z)$ is defined by $H(z) = \sum h(k)z^{-k}$. The poles of H are values of z for which $|H(z)|$ is infinite. An IIR filter is BIBO-stable iff its impulse response is absolutely summable, i.e., $\sum |h(k)|$ converges. For a BIBO-stable filter, for any $z \in \mathbb{C}$ such that $|z| \geq 1$, we have $|z^{-k}| \leq 1$; therefore, $|H(z)| \leq \sum |h(k)||z^{-k}| \leq \sum |h(k)|$, therefore, $|H(z)|$ is finite: z is not a pole of H . Therefore, for a BIBO-stable filter, any pole must be such that $|z| < 1$, i.e., within the unit circle. The other implication is also true, but we do not prove it here.

back loop(s) of the architecture. This, however, makes them very sensitive to any rounding of their coefficient.

Note that long-term memory can also be achieved using an FIR filter with a correspondingly large number of coefficients: a filter designer often has a choice between a compact but potentially unstable IIR filter and a stable but bulky FIR one. Apart from that, FIR filters are also widely used because they allow a strict linear phase response, a property that IIR filters can only approximate.

23.1.9 Filter Design: From a Frequency Specification to a Time-Domain Architecture

We now describe the classic filter design and implementation flow, available, for instance, in the popular Matlab `filterDesigner` tool (previously called `fdatool`) as illustrated by Fig. 23.15. It follows three *separate optimization steps*:

Filter design (FD) consists in finding a filter adhering to a frequency response. Such a filter is determined by the size parameters n and m , and the values of the coefficients a_i and b_i of the rational transfer function (23.19). For IIR filters, this is traditionally done by applying the bilinear transform to well-known continuous filters (e.g., Butterworth, Chebyshev). Modern approaches directly obtain the coefficients using optimization algorithms like the least- p th algorithm [Ant18, ch. 16]. For FIR filters, $a_i = 0$, therefore (23.19) becomes a polynomial whose coefficients b_i can be obtained using a Remez-type algorithm [Fil16] in the frequency domain⁶. Larger values of the filter order n allow for tighter transition bands and tighter band constraints, but at a higher architectural cost. Even for a given n , many different coefficient sets can fulfill a given frequency specification, offering a large design space.

Quantization (Q) consists in ensuring that the coefficients a_i and b_i are finitely representable (e.g., in fixed point), a prerequisite for a hardware implementation. Simply quantizing the coefficients obtained in the filter design step is likely to break the frequency response specification or even the stability. A reason for this is that, as already stated, most interesting IIR filters are close to being unstable (their poles are close to the unit circle). Several approaches have attempted to merge the FD and Q steps by only considering, in the FD step, filters with quantized coefficients [KO68; Kod12; VLH17; BFH18; VHL19]. Probabilistic error models can also be used [WK08].

⁶ Remez' algorithm can be used for IIR filters as well but was found to be problematic with respect to convergence and stability [BS75].

Implementation (I) consists in generating a valid hardware description using the quantized coefficients. The resulting hardware is no longer an LTI filter; it is not even linear due to the rounding errors. However, the implementation step can be performed in such a way that the hardware is last-bit accurate with respect to the filter it implements. This is the subject of the following section.

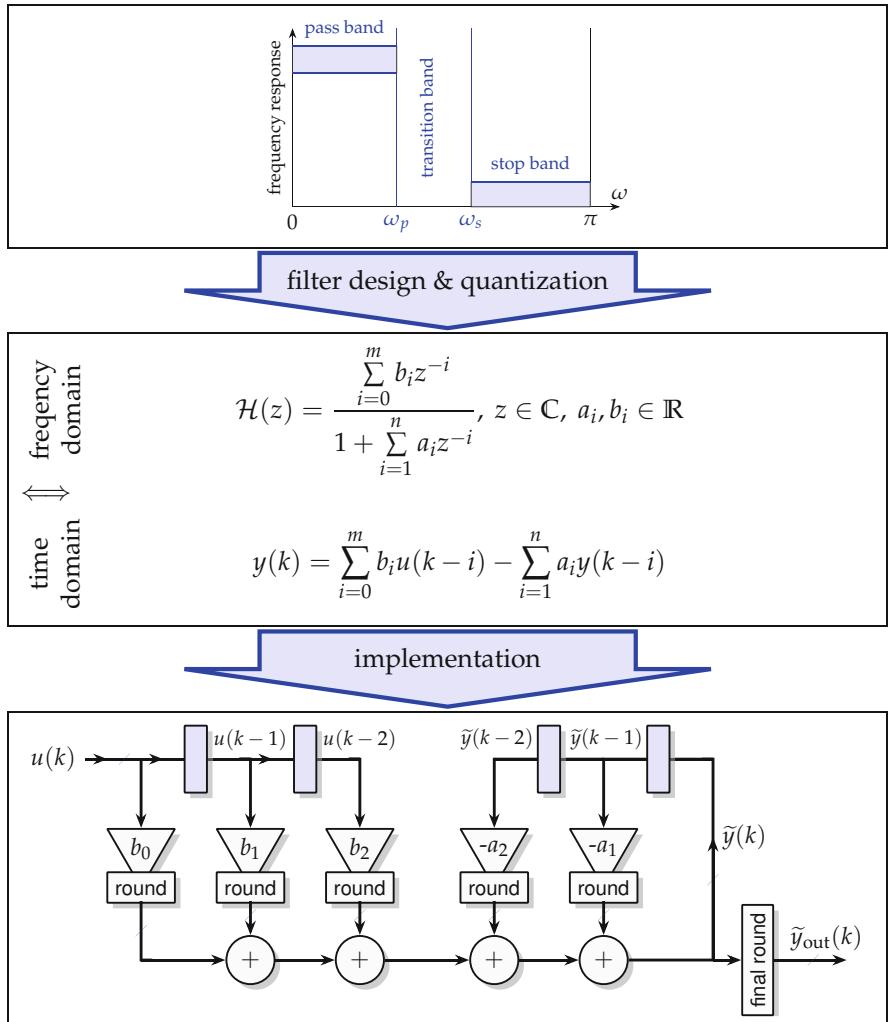


Fig. 23.15 From a frequency specification to an architecture.

23.2 An Arithmetic Approach to the Implementation of LTI Filters

In the implementation of a digital filter, all the values must belong to some finite-precision numbers. Figure 23.16 shows what happens when rounding is applied to a filter in the set \mathcal{L} of BIBO-stable LTI filters. There exists a significant subset of \mathcal{L} consisting of BIBO-stable LTI filters whose coefficients have a finite precision. However, rounding the coefficients of a filter in \mathcal{L} may well lead to an unstable filter, which therefore no longer belongs to \mathcal{L} .

A hardware digital filter must use quantized coefficients but is also restricted to only use quantized internal signals. Another point of view is that the outputs of operators (especially multipliers) must be rounded when recursions occur. Indeed, an unrounded multiplier by a nontrivial coefficient has more output bits than input bits and therefore cannot be used on an infinite recursive loop (the number of bits would go to infinity, which is not possible in a hardware circuit). Even for FIR filters without such loops, it is often desirable that the output number of bits is comparable to the number of input bits, not larger, which implies the rounding of multiplier outputs. But rounding is not linear; therefore, a digital circuit is rarely an LTI filter, as Fig. 23.16 illustrates. In this figure, there is a small set $\mathcal{L} \cap \mathcal{C}$ of hardware digital filters that are also stable LTI filters. Their coefficients must be finitely representable, and the hardware must evaluate (23.19) (or a mathematically equivalent formula) exactly, i.e., without any rounding. This limits this set to non-recursive filters (and a handful of trivial recursive filters) with more output bits than input bits, which is not typical in actual applications.

Outside of this small set, hardware digital filters involve signal rounding and are therefore not linear. An important consequence is that for most hardware digital filters, it is simply not possible to define a frequency response, a transfer function, or the poles of the latter: all these notions assume linearity. It is not surprising that hardware digital filters exhibit strange behaviors such as limit cycle oscillations (when the output of a hardware filter has an oscillating periodic behavior when it should converge to a constant value). Similarly, the question “does this hardware digital filter respect this frequency specification” is fundamentally ill-posed, since the frequency response of the hardware filter cannot be defined.

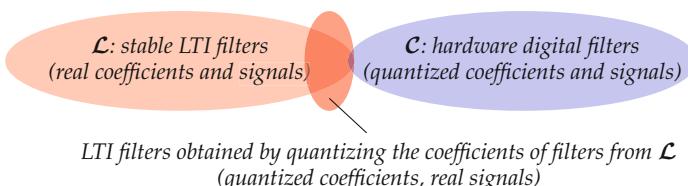


Fig. 23.16 Hardware digital filters are usually not BIBO-stable LTI filters.

Still, hardware digital filters are widely used, so at least some of them do work in practice. The sequel captures this formally using original arithmetic considerations.

Section 23.3 shows how to relate a hardware digital filter in \mathcal{C} to its ideal counterpart in \mathcal{L} by the same notion of faithfulness (or last-bit accuracy) that has been used in many other chapters of this book. It shows that this definition provides an effective solution to issues such as limit cycle oscillations. It also shows how to obtain a faithful implementation $\mathcal{C} \in \mathcal{C}$ of any stable LTI filter $\mathcal{H} \in \mathcal{L}$ from its (real) coefficients.

Section 23.4 extends this approach to the more interesting case of LTI filters designed from a frequency specification. We cannot say that a hardware digital filter respects a frequency specification, but we may say that it is *faithful* to a frequency specification as soon as it is faithful to one of the many LTI filters respecting this frequency specification. Again, hardware digital filters defined this way are as close as possible to their ideal counterparts. There remains the challenge to find the cheapest hardware digital filters faithful to the specification, and Sect. 23.4 also discusses this optimization problem.

23.3 Hardware Digital Filter Faithful to an LTI Filter

LTI filters faithful to their coefficients can be formalized using the same definition of faithfulness that has been used elsewhere in this book:

Definition 23.3 *A hardware digital filter \mathcal{C} is faithful to an LTI filter \mathcal{H} iff for any input signal, the difference between the output of \mathcal{C} and the output of \mathcal{H} is strictly less than one unit in the last place (ulp) of the output of \mathcal{C} .*

Chapter 3 gave some justifications of this choice of last-bit accuracy. This definition can be generalized to any finite-precision implementation, e.g., software, but the present book focuses on hardware digital filters. Note that technically, \mathcal{C} will be the implementation of some filter structure, but we do not need to mention this filter structure when referring to the filter \mathcal{H} , since \mathcal{H} can be realized by many filter structures which are all mathematically equivalent (e.g., the four architectures of Figs. 23.9 and 23.11).

This faithfulness specification has many useful practical consequences.

1. Designers are mostly interested in stable LTI filters. If the filter \mathcal{H} is BIBO-stable, then any hardware digital filter \mathcal{C} faithful to \mathcal{H} according to Definition 23.3 will also be BIBO-stable.
2. Definition 23.3 does not make the circuits linear, but it does ensure that nonlinearities are harmless in practice. For instance, if the ideal filter \mathcal{H} is stable, then its impulse response converges to a finite value. The amplitude of possible limit cycle oscillations in \mathcal{C} around this value will be less than one ulp: it belongs to the last-bit noise that users of hardware must live with.

3. All faithful implementations are functionally equivalent in practice. Therefore, the choice of a hardware digital filter becomes an optimization problem among functionally equivalent implementations which can be compared quantitatively on other metrics, e.g., on their cost. This comparison can be performed by automatic tools that explore possible implementations, some of which will be reviewed in Sect. 23.4.3.

Section 23.4 will extend this definition to a filter faithful to a frequency specification, with the same benefits. Before that, it is necessary to introduce the mathematical tool that quantifies BIBO stability: the worst-case peak gain.

23.3.1 Worst-Case Peak Gain of an LTI Filter

Definition 23.4 (Peak value of a signal) For a bounded signal x , the peak value of x is denoted as $\|x\|_\infty$ and is defined as $\|x\|_\infty = \max_k |x(k)|$.

Definition 23.5 (Worst-case peak gain) For a BIBO-stable LTI filter \mathcal{H} , the Worst-Case Peak-Gain (WCPG), denoted $\langle\langle \mathcal{H} \rangle\rangle$, is defined as the largest peak value of the output y over all possible input signals u whose peak value is 1:

$$\langle\langle \mathcal{H} \rangle\rangle = \max_{\|u\|_\infty=1} \|y\|_\infty \quad (23.20)$$

To approach this worst case, remember that $y(k) = \sum_{i=0}^k h(i)u(k-i)$ where h is the impulse response of \mathcal{H} . If k is finite, this sum is maximized (among signals of peak value 1) for the signal u defined as

$$u(k-i) = \begin{cases} 1 & \text{if } h(i) > 0 \\ -1 & \text{if } h(i) < 0 \end{cases} \quad (23.21)$$

which yields $y(k) = \sum_{i=0}^k |h(i)|$. Note that this sum grows with k . Eventually, the WCPG can be computed as

$$\langle\langle \mathcal{H} \rangle\rangle = \sum_{k=0}^{\infty} |h(k)| \quad (23.22)$$

and indeed this sum converges for BIBO-stable filters. In addition, (23.21) shows how to build a signal whose response comes arbitrarily close to the WCPG. Note that this signal has a very small probability to appear in a practical situation: the WCPG is a pessimistic measure compared to probabilistic approaches [WK08]. However, it does provide a guarantee.

By linearity, for any BIBO-stable LTI filter \mathcal{H} with input u , a tight bound on the peak value of its output y is given by

$$\bar{y} = \langle\langle \mathcal{H} \rangle\rangle \bar{u}. \quad (23.23)$$

This definition of the WCPG can be generalized to MIMO filters [BB92; VHL19]. FloPoCo uses a library⁷ that computes safe bounds of WCPGs with arbitrary precision [VHL19]. The test bench generators in FloPoCo implementations of IIR filter also build worst-case signals implementing (23.21) to test the resulting architectures.

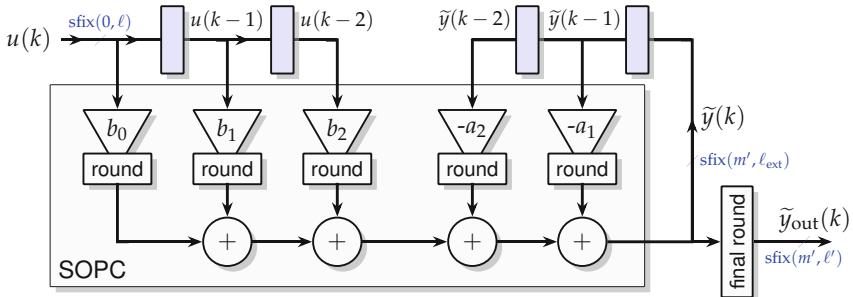


Fig. 23.17 Direct-form implementation of an IIR filter using a SOPC component.

The sequel of this section details the construction of a faithful filter implementing the direct form of Fig. 23.17. Beyond this detailed example, the techniques presented there can be used for other types of implementations.

23.3.2 Interface Considerations

Let us first detail some of the parameters appearing in Fig. 23.17. Since the considered filters are linear, we may assume without loss of generality that $\bar{u} = 1$, i.e., hence the position of the input most significant bit (MSB) is 0. The input format is then defined by the position ℓ of its least significant bit (LSB). The output format is defined by its LSB ℓ' that specifies the expected accuracy of the filter.

The output MSB m' is usually part of the application specification. Based on (23.23), an upper bound on m' of the output y is defined by $m' \leq \lceil \log_2 \langle\langle \mathcal{H} \rangle\rangle \rceil$. Technically, it may happen that rounding errors propagate all the way to the MSB. In a faithful filter, these errors will be bounded by

⁷ <https://github.com/fixif/WCPG.git>

$2^{\ell'}$. Therefore, an upper bound on the output MSB, taking into account the rounding error, is

$$m' \leq \lceil \log_2 (\langle\langle \mathcal{H} \rangle\rangle + 2^{\ell'}) \rceil . \quad (23.24)$$

This bound is often extremely pessimistic, and application-level consideration may allow for a value of m' that is much smaller. Consider, for instance, the bell model using multimodal additive sound synthesis (Fig. 23.12, p. 682). Some of the IIR filters on this figure have a WCPG of almost 2×10^5 , which would mean $m' = 18$ bits. However, the typical input to these IIR is not a random signal, but a sequence of distant impulses modeling the hammer of the bell: for this particular application, it makes more sense to define m' to match the peak of the actual impulse response as introduced with Definition 23.4.

The architecture of Fig. 23.17 internally uses a fixed-point format that is wider than the output format. The sequel will show how to determine the LSB of this format ℓ_{ext} . The MSB position of this internal format also m' . Indeed, even if overflows occur during the internal computations, the final result will be correct since fixed-point computations are performed modulo $2^{m'}$.

Considering all this, a minimal interface to an IIR architecture generator inputting real (non-quantized) coefficients is shown in Fig. 23.18. This interface does not depend at all on the IIR structure, but the internal parameters, such as ℓ_{ext} , obviously do.

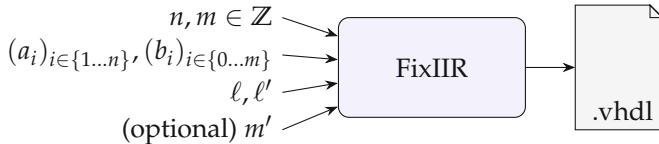


Fig. 23.18 Black-box interface to a generator of faithful IIR architectures from real coefficients.

23.3.3 Overall Error Analysis of the Implementation of a Direct-Form LTI Filter

In Fig. 23.17, the SOPC block inputs the $u(k)$ and the $\tilde{y}(k)$. The input signal $u(k)$ can be considered exact, in the sense that whatever error it may carry is not due to the filter under consideration. Conversely, the internal signal $\tilde{y}(k)$ differs from the ideal $y(k)$ computed by the ideal filter. Let us define

$\delta_t(k)$ as the error of $\tilde{y}(k)$ with respect to $y(k)$:

$$\delta_t(k) = \tilde{y}(k) - y(k). \quad (23.25)$$

This error is potentially amplified by the feedback loop of the architecture, but this can be controlled by using an extended precision ℓ_{ext} for $\tilde{y}(k)$.

The final output $\tilde{y}_{\text{out}}(k)$ will be obtained by rounding this intermediate value $\tilde{y}(k)$ in the **final round** box of Fig. 23.17. The corresponding error is

$$\delta_{\text{final round}}(k) = \tilde{y}_{\text{out}}(k) - \tilde{y}(k) . \quad (23.26)$$

The overall evaluation error is defined as

$$\delta_{\text{total}}(k) = \tilde{y}_{\text{out}}(k) - y(k) \quad (23.27)$$

$$= \tilde{y}_{\text{out}}(k) - \tilde{y}(k) + \tilde{y}(k) - y(k) \quad (23.28)$$

$$= \delta_{\text{final round}}(k) + \delta_t(k) . \quad (23.29)$$

The issue is now to relate $\delta_t(k)$ to the rounding errors due to the SOPC. Since it computes an approximation to $\sum b_i u(k-i) - \sum a_i \tilde{y}(k-i)$, all these errors can be summarized as a single term:

$$\delta_r(k) = \tilde{y}(k) - \left(\sum_{i=0}^m b_i u(k-i) - \sum_{i=1}^n a_i \tilde{y}(k-i) \right) . \quad (23.30)$$

This error $\delta_r(k)$ measures how much a result $\tilde{y}(k)$ computed by the SOPC architecture diverges from that computed by an ideal SOPC that would use the infinitely accurate coefficients a_i and b_i , and be free of rounding errors, but would nevertheless input the same (finite precision) $u(k-i)$ and $\tilde{y}(k-i)$ as the hardware digital filter. This error depends on the SOPC implementation. What is important is that it may be bound using the techniques presented in Sect. 12.5. It may be made arbitrarily small by using an arbitrarily small value of ℓ_{ext} . In the case when the coefficients a_i and b_i are finitely representable (i.e., they have been quantized), it may even be exact ($\delta_r(k) = 0$) if the internal format is accurate enough.

23.3.4 Error Amplification in the Feedback Loop

Let us rewrite, in the right-hand side of (23.30) by inserting (23.25):

$$\begin{aligned}
\delta_r(k) &= \tilde{y}(k) - \sum_{i=0}^m b_i u(k-i) + \sum_{i=1}^n a_i y(k-i) \\
&\quad + \sum_{i=1}^n a_i \delta_t(k-i) \\
&= \tilde{y}(k) - y(k) + \sum_{i=1}^n a_i \delta_t(k-i) \quad (\text{using (23.15)}) \\
&= \delta_t(k) + \sum_{i=1}^n a_i \delta_t(k-i) \quad (\text{using (23.25)}). \tag{23.31}
\end{aligned}$$

By rewriting Eq. (23.31) as

$$\delta_t(k) = \delta_r(k) - \sum_{i=1}^n a_i \delta_t(k-i) \tag{23.32}$$

we obtain the equation of an IIR filter inputting $\delta_r(k)$ and outputting $\delta_t(k)$, whose transfer function is

$$\mathcal{H}_\delta(z) = \frac{1}{1 + \sum_{i=1}^n a_i z^{-i}}. \tag{23.33}$$

Figure 23.19 illustrates this relationship between the ideal output y , the implemented output \tilde{y}_{out} and the three error terms.

The Worst-Case Peak-Gain theorem may now be applied to \mathcal{H}_δ with input δ_r , bounding δ_t by

$$\bar{\delta}_t = \langle\langle \mathcal{H}_\delta \rangle\rangle \bar{\delta}_r. \tag{23.34}$$

Therefore, $\bar{\delta}_t$ can be kept as low as needed by increasing the internal precision ℓ_{ext} to reduce $\bar{\delta}_r$.

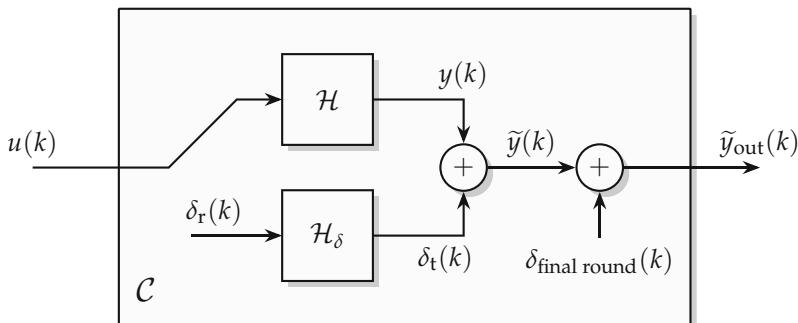


Fig. 23.19 Model of a hardware digital filter \mathcal{C} faithful to the ideal filter \mathcal{H} .

23.3.5 Putting All Together

Finally, (23.29) can be rewritten as

$$\bar{\delta}_{\text{total}} = \bar{\delta}_{\text{final round}} + \langle\langle \mathcal{H}_\delta \rangle\rangle \bar{\delta}_r. \quad (23.35)$$

The objective of last-bit accuracy of the architecture translates into the constraint $\bar{\delta}_{\text{total}} < 2^{\ell'}$. Taking into account the final rounding (with error bound $\bar{\delta}_{\text{final round}} = 2^{\ell'-1}$), we obtain the constraint on the error $\bar{\delta}_r$ of the SOPCs:

$$\bar{\delta}_r < \frac{2^{\ell'-1}}{\langle\langle \mathcal{H}_\delta \rangle\rangle}. \quad (23.36)$$

This constraint may finally be translated to a constraint on the LSB ℓ_{ext} of the intermediate result. For instance, a faithful SOPC built as per Sect. 12.5.5 ensures $\bar{\delta}_r < 2^{\ell_{\text{ext}}}$. The optimal value of ℓ_{ext} that ensures this constraint may be deduced from (23.36) as

$$\ell_{\text{ext}} = \ell' - 1 - \lceil \log_2 \langle\langle \mathcal{H}_\delta \rangle\rangle \rceil. \quad (23.37)$$

In other words, the internal format adds $1 + \lceil \log_2 \langle\langle \mathcal{H}_\delta \rangle\rangle \rceil$ LSB guard bits to the output format. In FloPoCo, the implementation of this error analysis actually uses a guaranteed overestimation of $\langle\langle \mathcal{H}_\delta \rangle\rangle$ [VHL19]. This ensures that rounding errors in the computation of $\langle\langle \mathcal{H}_\delta \rangle\rangle$ itself do not jeopardize the accuracy. Because of this overestimation, it is in principle possible (but extremely rare) that FloPoCo computes on one bit more than what was required by (23.37).

Hands on: FixIIR

On the bell model described by Fig. 23.12, the resonating filter with the lowest frequency is defined by $a_1^1 = -1.99510896$ and $a_2^1 = 0.999985754$. The poles of this filter are obviously close to the unit circle.

A guaranteed faithful implementation of this filter can be synthesized by the following command:

```
flopoco FixIIR lsbin=-8 lsbout=-8 \
coeffb="1:0:-1" coefffa="-1.99510896:0.999985754"
```

On this example, FloPoCo reports (after several seconds of computation) $\langle\langle \mathcal{H} \rangle\rangle = 178,750$ and $\langle\langle \mathcal{H}_\delta \rangle\rangle = 1.2806e+06$. This means that there exists an exciting signal $x(k)$ such that the output amplitude can be 178,750 times the input amplitude. However, for a unit impulse signal, it is easy to show that the output does not exceed 2 in absolute value. For this reason, FixIIR offers the possibility to override the application-specific worst-case peak gain values.

23.4 Hardware Digital Filter Faithful to a Frequency Specification

We may now address the construction of hardware digital filters out of a frequency specification.

23.4.1 Formal Definition

Definition 23.6 A hardware filter $\mathcal{C} \in \mathcal{C}$ is faithful to a frequency specification iff there exists a BIBO-stable LTI filter $\mathcal{H} \in \mathcal{L}$ such that

- \mathcal{H} fulfills the frequency specification, and
- \mathcal{C} is faithful to \mathcal{H} (in the sense of Definition 23.3).

This definition (which is, to our knowledge, original) has several advantages.

Firstly, it ensures that nonlinearities are harmless in practice, thanks to Definition 23.3. They won't entail instability, and possible limit cycle oscillations will be limited to one ulp of the result.

Secondly, Definition 23.6 enables the minimalistic interface to a filter design tool depicted in Fig. 23.20. Tools such as Matlab's `filterDesigner` expose many other parameters: filter order (n and m), the quantization parameters of the a_i and b_i , the values of these coefficients, the format of $\tilde{y}(k)$, and the formats of other internal signals, etc. A wrong choice of these parameters may entail an unstable implementation. We claim that most of these parameters are implementation technicalities that should be determined automatically. There should only remain a handful of knobs controlling, e.g., cost/performance trade-offs.

Finally, with Definition 23.6, hardware filter design can be expressed as a simple global optimization problem: find the best circuit (according to some metric) faithful to a given frequency specification.

The remainder of this chapter addresses this optimization problem, building upon existing literature that has formalized and solved relevant subproblems.

23.4.2 Hardware Filter Design as an Optimization Problem

In general, an optimization problem is defined by a parameter space, a set of constraints, and an objective function based on cost and performance metrics.

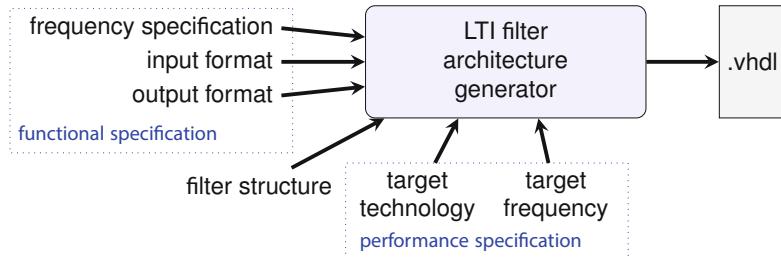


Fig. 23.20 A simple interface to a hardware filter design tool.

For filters, the **parameter space** includes the filter order (n and m), and the coefficients (a_i and b_i), quantized or not. These are the main functional parameters.

Among the performance parameters, two are of special importance. The first one is the arithmetic to be used internally, in particular for constant multiplications. Chapter 12 has reviewed the two main families of methods for this. Each method can exploit the sharing of intermediate results to improve area, performance, or power. This defines a wide spectrum of low-level filter realizations whose cost depends on the coefficient values in a nontrivial way.

The second important parameter is the *filter structure*: for both FIR and IIR filters, there is a wide range of possible realizations [Ant05], a few of which have been reviewed in Sect. 23.1.5. Each realization corresponds to some algebraic rewriting of (23.19), and the direct form of Fig. 23.17 is among the simplest. Other well-studied realizations include lattice wave filters [Fet86], filter cascades [MW15], etc. A long-term purpose here would be to improve the qualitative and experimental know-how of filter designers (“this structure is more stable than that one in this context”) with quantitative assessments of the cost of a stable-by-design implementation using each candidate structure. Here the filter structure is seen as a performance parameter, but it can also be part of the functional specification (e.g., some structures ensure linear phase).

Among the **constraints** of the problem, there are frequency-domain constraints capturing the frequency specification, and time-domain constraints capturing the last-bit accuracy specification of Definition 23.6. Frequency-domain constraints have to be embedded into the optimization as this has been done for optimizing FIR filters using ILP in [KVF22]. We will review these ILP constraints below. The obtained optimization result may be checked using safe arbitrary-precision techniques [Fil16; VHL19] as a protection against the inevitable rounding errors in the FD+Q software. Time-domain accuracy constraints may be managed as per previous section.

The **cost functions** may include high-level metrics, typically the number of adders in a shift-and-add filter. However, considering the rounded inter-

mediate results, all the adders are not of the same size. Finer metrics may take this into account, with accurate gate-level models of area, performance, and power consumption [JGW05; LYM15; YLY17].

23.4.3 State of the Art in the Design of Filters Faithful to a Frequency Specification

The literature is solving increasingly relevant subproblems, but to our knowledge, a tool addressing fully the optimization problem stated above is still missing.

The combination of the FD + Q steps (as defined in Sect. 23.1.9 above) has been studied since the 1960s [KO68] and can even be regarded as solved for certain practical instances of fixed-point FIR design [Kod12; BFH18].

Section 23.3 has described how to solve the combined Q+I optimization problem, and the FloPoCo implementation [Vol+19] addresses a bit-level cost model, albeit limited to table-based constant multipliers and direct form I.

A large body of work exists for the I step alone. Some methods explore filter structures: the structural adders in FIR filters can be rearranged to reduce their word size [Che+17]; a FIR filter may be factored into a cascade of subsections [MW15]. Several approaches to the optimization of the internal word sizes exist [CCL04, Ch. 4]. Other hardware optimizations in the I step target the MCM problem, using shift-and-add [VP07; Gus07; Aks+08; Kum18] or precomputed tables [Cha94; FC11] (see Sect. 12.5).

However, the result of these optimizations in the I step depends strongly (and in a non-monotonous way) on the coefficient values, which are fixed in the earlier FD and Q steps. With Definition 23.6, we may define a joint optimization of the FD+Q+I steps, for instance, finding the set of fixed-point coefficients that minimize the overall cost among MCM-based architectures faithful to a frequency specification.

We are almost there for FIR filters (in direct or transposed form), either using a custom branch and bound algorithm [AFM15] or integer linear programming (ILP) [KVF22]. Another combined FD+Q+I optimization for FIR filters focuses on power consumption [YLY17], by searching for shift-and-add solutions with low adder depth using a combination of branch and bound and linear programming. However, so far, these works only find filters in $\mathcal{L} \cap \mathcal{C}$ thus cannot capture filters where the output is rounded.

For IIR filters, a global FD+Q+I optimization has been formalized [Gar+22], but this model is currently limited to second-order filters and does not use a bit-level cost function for the shift-and-add trees.

We conclude this chapter with a case study, the combined FD+Q+I steps for linear phase FIR filters using ILP [KVF22], to show how frequency

constraints can be integrated in the MCM ILP formulation presented in Sect. 12.5.1.1.

23.4.4 Frequency Constraints for Linear-Phase FIR Filters in ILP

The implementation of a FIR filters is essentially an MCM, but here the coefficients themselves have to be determined. It is clear from Definition 23.6 that several coefficient sets may fulfill the same frequency specification, and the purpose of this section is to build an ILP formulation that selects the coefficient set whose shift-and-add MCM implementation minimizes the overall number of adders (counting adders from the constant multipliers as well as structural adders).

The focus of this case study is linear-phase FIR filters. First, because linear-phase FIR filters are desired in many applications [OS09]. Second, they are also interesting because of the symmetry in their coefficients, which allows that only about half of the multiplications have to be computed. Finally, their frequency response is real-valued (and not complex-valued as in the general case), which simplifies its integration into ILP. Let us detail on the latter.

A linear phase FIR filter of n -th order can be described with the zero-phase frequency response [Ant05]

$$H_R(\omega) = \sum_{i=0}^{m-1} h_i c_i(\omega), \quad \omega \in [0, \pi] \quad (23.38)$$

where $c_i(\omega)$ is a trigonometric function given in Table 23.1 and m denotes the number of coefficients after removing identical or negated ones due to symmetry. As Table 23.1 shows, both m and $c_i(\omega)$ depend on the filter symmetry and on the parity of n .

The magnitude of the real-valued zero-phase frequency response is identical to that of the complex-valued transfer function of the filter, i.e.,

$$|H(e^{j\omega})| = |H_R(\omega)|. \quad (23.39)$$

Hence, for a frequency specification defined by the lower bound $\underline{D}(\omega)$ and upper bound $\overline{D}(\omega)$ of the output frequency response, a valid frequency response has to fulfill the constraint

$$\underline{D}(\omega) \leq H_R(\omega) \leq \overline{D}(\omega) \quad \forall \omega \in [0, \pi] \quad (23.40)$$

An issue is that $H_R(\omega)$ depends on $c_i(\omega)$, which is not linear and infinite. The constraint is therefore evaluated for a discrete subset of values: $\omega \in$

Table 23.1 Relation between filter order n , number of coefficients m and function $c_i(\omega)$ for different filter types.

Type	Symmetry	n	m	$c_i(\omega)$
I	Symmetric	Even	$\frac{n}{2} + 1$	$c_i(\omega) = \begin{cases} 1 & \text{for } m = 0 \\ 2 \cos(\omega m) & \text{for } m > 0 \end{cases}$
II	Symmetric	Odd	$\frac{n+1}{2}$	$c_i(\omega) = 2 \cos(\omega(m + 1/2))$
III	Antisymmetric	Even	$\frac{n}{2}$	$c_i(\omega) = 2 \sin(\omega(m - 1))$
IV	Antisymmetric	Odd	$\frac{n+1}{2}$	$c_i(\omega) = 2 \sin(\omega(m + 1/2))$

$\Omega_d \subset [0, \pi]$. A uniform discretization of $16m$ values in Ω_d typically proves sufficiently dense [Kod99].

Substituting (23.38) in (23.40) provides a finite set of linear constraints. Their real coefficients h_m are then replaced with finite-precision scaled integers:

$$h_m = 2^\ell C_m \quad (23.41)$$

where ℓ is the LSB position of the coefficients (a parameter that has to be fixed in advance and can be explored outside of the ILP). These constraints may now be integrated in the MCM ILP formulation discussed in Sect. 12.5.1.1. Each C_m is no longer a constant, but a variable.

It is also possible to consider a variable gain and to include the count of structural adders in the objective function (some constants may be zeroes, as illustrated by the example below) [KVF22] but we do not detail this here.

To give an impression about the optimization potential, let us consider an example low-pass filter. The filter should pass low frequencies up to $\omega_p = 0.3\pi$ with an error (passband ripple) of not more than $\delta_p = 0.0157$ (i.e., $\overline{D}(\omega)$ and $\underline{D}(\omega)$ are 1 ± 0.0157 , respectively) and should attenuate from $\omega_s = 0.5\pi$ by at least -43.6 dB which is a factor of 0.0066 (i.e., $\overline{D}(\omega) = 0.0066$).

A naive filter design in Matlab would first use Remez algorithm, followed by a quantization. Here a filter with 25 coefficients is required, and they can be quantized to 12 bits without violating the filter specification. The quantized impulse response is $h_{\text{naive}} = 2^{-12} \cdot (5, 30, 1, -53, -49, 61, 141, 4, -261, -236, 364, 1229, 1643, 1229, 364, -236, -261, 4, 141, 61, -49, -53, 1, 30, 5)$ and Fig. 23.21a shows its frequency response. Due to the symmetries, 13 multiplications are necessary to perform from which two are trivial (coefficient 1 and coefficient 4 = 2^2). Using MCM techniques as introduced in Sect. 12.5, the remaining 11 multiplications can be reduced to only 13 additions. As we need 24 additions to sum up the delayed products (the structural adders), the total filter requires $13 + 24 = 37$ additions. Figure 23.22a shows the resulting filter.

Directly designing the FIR filter for minimal adders as presented in the ILP formulation above results in a filter with 24 coefficients on 10 bits, with the impulse response $h_{\text{opt}} = 2^{-10} \cdot (3, 2, -6, -12, 0, 20, 16, -24, -52, 0, 136, 256, 136, 0, -52, -24, 16, 20, 0, -12, -6, 2, 3)$. Figure 23.21b illustrates that the frequency specification (drawn in black) is still met, while Fig. 23.22b shows that the multiplications can be performed by just 4 additions. In addition, four structural adders can be saved by coefficients that are zero. In total, we have $4 + 19 = 23$ additions, which are significantly less compared to the 37 additions of the naive approach.

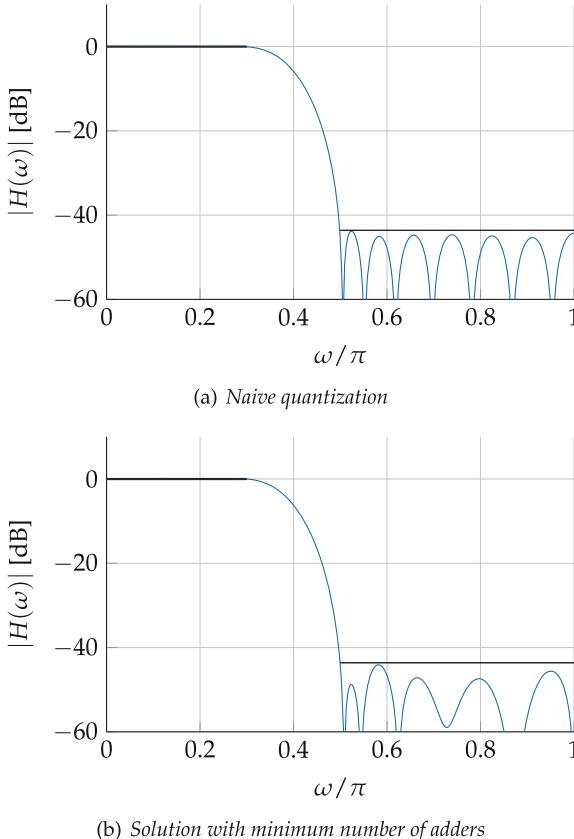


Fig. 23.21 Frequency specification and response.

This section showed how frequency constraints could be merged into the ILP of a MCM and the potential of this approach. However, it does not yet provide a perfect solution. The filter depicted in Fig. 23.22 is an exact one (no rounding on the datapaths). If it inputs, say, 8-bit data, it outputs an exact result on $8 + 10 = 18$ bit. On Fig. 23.16, such a filter belongs to the small

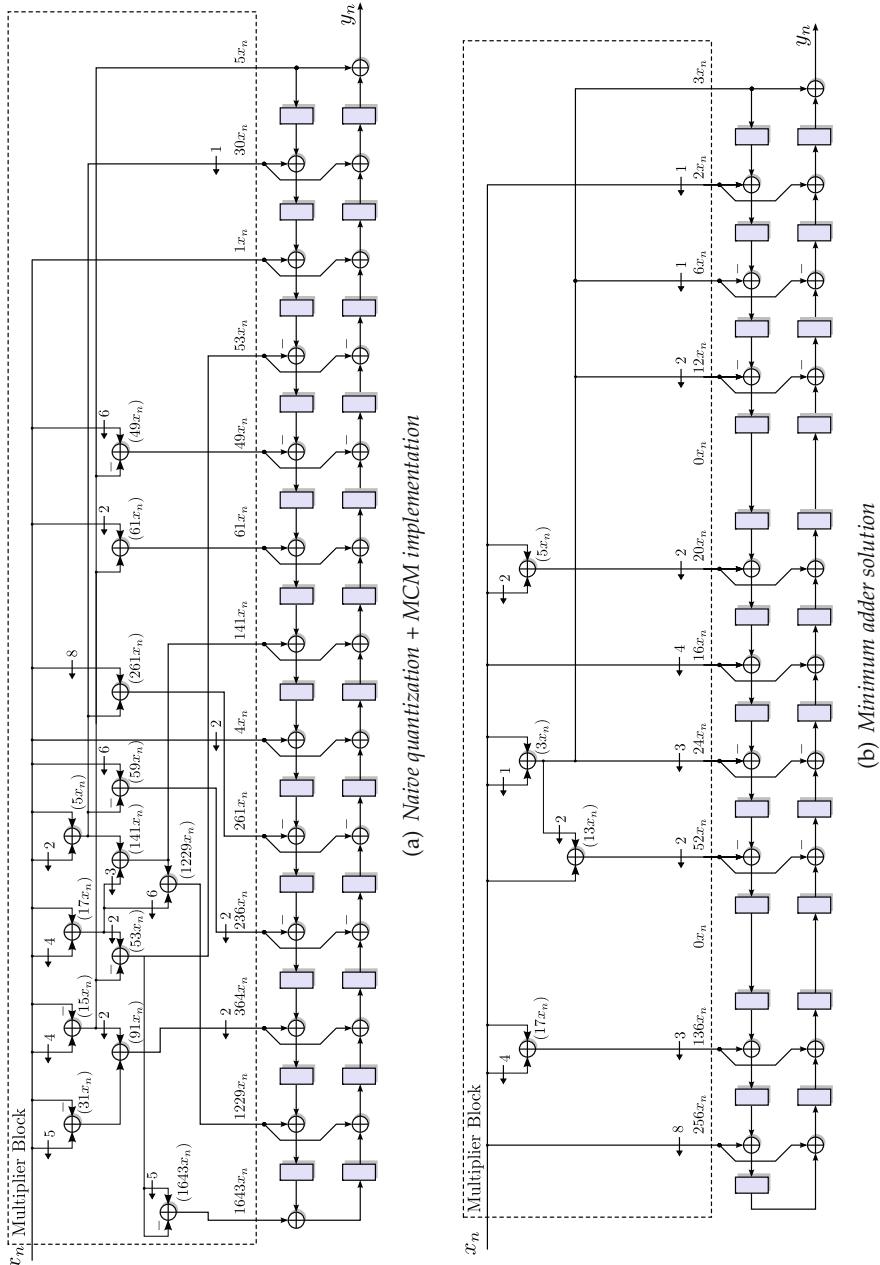


Fig. 23.22 Comparison of the naive and ILP approaches to filter design, quantization, and implementation.

intersection $\mathcal{L} \cap \mathcal{C}$. Another point of view is that this approach does not allow for a specification of the output format and thus does not completely address the interface of Fig. 23.20.

The good news is that a faithful (and even correct) rounding to, say, 8 bits of this exact result is easy to implement: we do have a recipe for implementing this interface and obtaining good circuits faithful to a frequency specification [Din+19; GVK22; GV23]. The bad news is that this approach cannot claim the optimality of the resulting circuit (among all possible circuits faithful to a frequency specification).

To achieve optimality, truncations have to be considered in the MCM problem, which also means counting full adders, not adders. This is possible [GVK22], but at the time of writing this book, a single global ILP formulation of the FD+Q+I remains to be written.

Beyond this, it remains an open problem to address recursive filters of higher order, or other complex filter structures like cascades. Solving the FD+Q+I steps simultaneously for such state-of-the-art filter structures is an exciting research problem for the years to come.

References

- [802.15.4] *IEEE Standard for Information technology– Telecommunications and information exchange between systems– Local and metropolitan area networks– Specific requirements– Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)*. 2006. (cit. on p. 682).
- [AFM15] Levent Aksoy, Paulo Flores, and José Monteiro. “Exact and Approximate Algorithms for the Filter Design Optimization Problem”. In: *Signal Processing, IEEE Transactions on* 63.1 (2015), pp. 142–154. (cit. on p. 697).
- [Aks+08] Levent Aksoy, Eduardo da Costa, Paulo Flores, and José Monteiro. “Exact and Approximate Algorithms for the Optimization of Area and Delay in Multiple Constant Multiplications”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.6 (2008), pp. 1013–1026. (cit. on p. 697).
- [Ant05] Andreas Antoniou. *Digital Signal Processing: Signals, Systems, and Filters*. McGraw-Hill Education, 2005. (cit. on pp. 670, 672, 696, 698).
- [Ant18] Andreas Antoniou. *Digital Filters: Analysis, Design, and Signal Processing Applications*. McGraw-Hill Education, 2018. (cit. on pp. 679, 685).
- [BB92] V. Balakrishnan and S. Boyd. “On Computing the Worst-Case Peak Gain of Linear Systems”. In: *Systems & Control Letters* 19.2 (1992), pp. 265–269. (cit. on p. 690).

- [BFH18] Nicolas Brisebarre, Silviu-Ioan Filip, and Guillaume Hanrot. "A Lattice Basis Reduction Approach for the Design of Finite Wordlength FIR Filters". In: *IEEE Transactions on Signal Processing* 66.10 (2018), pp. 2673–2684. (cit. on pp. [685](#), [697](#)).
- [BS75] F. Brophy and A. Salazar. "Synthesis of Spectrum Shaping Digital Filters of Recursive Design". In: *IEEE Transactions on Circuits and Systems* 22.3 (1975), pp. 197–204. (cit. on p. [685](#)).
- [CCL04] George A. Constantinides, Peter Y.K. Cheung, and Wayne Luk. *Synthesis and optimization of DSP algorithms*. Kluwer, 2004. (cit. on p. [697](#)).
- [Cha94] Ken D. Chapman. "Fast Integer Multipliers Fit in FPGAs". In: *Electronic Design News* (1994). (cit. on p. [697](#)).
- [Che+17] Jiajia Chen, Chip-Hong Chang, Jiatao Ding, Rui Qiao, and Mathias Faust. "Tap Delay-and-Accumulate Cost Aware Coefficient Synthesis Algorithm for the Design of Area-Power Efficient FIR Filters". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 65.2 (2017), pp. 712–722. (cit. on p. [697](#)).
- [Din+19] Florent de Dinechin, Silviu-Ioan Filip, Luc Forget, and Martin Kumm. "Table-Based versus Shift-And-Add Constant Multipliers for FPGAs". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2019. (cit. on p. [702](#)).
- [DM95] Debjit Das Sarma and David W. Matula. "Faithful Bipartite ROM Reciprocal Tables". In: *12th Symposium on Computer Arithmetic*. Ed. by S. Knowles and W.H. McAllister. IEEE, 1995, pp. 17–28. (cit. on p. [669](#)).
- [FC11] Mathias Faust and Chip-Hong Chang. "Bit-parallel Multiple Constant Multiplication using Look-Up Tables on FPGA". In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2011, pp. 657–660. (cit. on p. [697](#)).
- [Fet86] Alfred Fettweis. "Wave Digital Filters: Theory and Practice". In: *Proceedings of the IEEE* 74.2 (1986), pp. 270–327. (cit. on p. [696](#)).
- [Fil16] Silviu-Ioan Filip. "A Robust and Scalable Implementation of the Parks-McClellan Algorithm for Designing FIR Filters". In: *ACM Transactions on Mathematical Software* 43.1 (2016), 7:1–7:24. (cit. on pp. [685](#), [696](#)).
- [Gar+22] Rémi Garcia, Anastasia Volkova, Martin Kumm, Alexandre Goldsztejn, and Jonas Kuhle. "Hardware-aware Design of Multiplierless Second-Order IIR Filters with Minimum Adders". In: *IEEE Transactions on Signal Processing* 70 (2022), pp. 1673–1686. ISSN: 1053-587X. (cit. on p. [697](#)).
- [Gus07] Oscar Gustafsson. "A Difference Based Adder Graph Heuristic for Multiple Constant Multiplication Problems". In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2007, pp. 1097–1100. (cit. on p. [697](#)).

- [GV23] Remi Garcia and Anastasia Volkova. "Toward the Multiple Constant Multiplication at Minimal Hardware Cost". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* (2023), pp. 1–13. (cit. on p. [702](#)).
- [GVK22] Rémi Garcia, Anastasia Volkova, and Martin Kumm. "Truncated Multiple Constant Multiplication with Minimal Number of Full Adders". In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2022, pp. 263–267. (cit. on p. [702](#)).
- [JGW05] Kenny Johansson, Oscar Gustafsson, and Lars Wanhammar. "A Detailed Complexity Model for Multiple Constant Multiplication and an Algorithm to Minimize the Complexity". In: *European Conference on Circuit Theory and Design*. Vol. 3. 2005, III/465–III/468 vol. 3. (cit. on p. [697](#)).
- [KO68] J. Knowles and E. Olcayto. "Coefficient Accuracy and Digital Filter Response". In: *IEEE Transactions on Circuit Theory* 15.1 (1968), pp. 31–41. (cit. on pp. [685](#), [697](#)).
- [Kod12] Dušan M. Kodek. "LLL Algorithm and the Optimal Finite Wordlength FIR Design". In: *IEEE Transactions on Signal Processing* 60.3 (2012), pp. 1493–1498. (cit. on pp. [685](#), [697](#)).
- [Kod99] Dušan M. Kodek. "Design of Optimal Finite Wordlength FIR Digital Filters". In: *European Conference on Circuit Theory and Design (ECCTD)*. 1999, pp. 401–404. (cit. on p. [699](#)).
- [Kum18] Martin Kumm. "Optimal Constant Multiplication using Integer Linear Programming". In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2018. (cit. on p. [697](#)).
- [KVF22] Martin Kumm, Anastasia Volkova, and Silviu-Ioan Filip. "Design of Optimal Multiplierless FIR Filters with Minimal Number of Adders". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022). (cit. on pp. [696](#), [697](#), [699](#)).
- [LYM15] Xin Lou, Ya Jun Yu, and Pramod Kumar Meher. "Fine-Grained Critical Path Analysis and Optimization for Area-Time Efficient Realization of Multiple Constant Multiplications". In: *IEEE Transactions on Circuits and Systems I* 62.3 (2015), pp. 863–872. (cit. on p. [697](#)).
- [MW15] Alireza Mehrnia and Alan N. Willson. "Optimal Factoring of FIR Filters". In: *IEEE Transactions on Signal Processing* 63.3 (2015), pp. 647–661. (cit. on pp. [684](#), [696](#), [697](#)).
- [OS09] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing*. 3rd ed. Prentice Hall Press, 2009. (cit. on pp. [670](#), [679](#), [683](#), [684](#), [698](#)).
- [Smi22] Julius Orion Smith. *Introduction to digital filters with audio applications*. 2022. URL: <http://ccrma.stanford.edu/~jos/filters/>. (cit. on pp. [670](#), [682](#)).
- [Sun+84] David A. Sunderland, Roger A. Strauch, Steven S. Wharfield, Henry T. Peterson, and Christopher R. Role. "CMOS/SOS Fre-

- quency Synthesizer LSI Circuit for Spread Spectrum Communications". In: *IEEE Journal of Solid-State Circuits* 19.4 (1984), pp. 497–506. (cit. on p. 669).
- [VHL19] Anastasia Volkova, Thibault Hilaire, and Christoph Lauter. "Arithmetic Approaches for Rigorous Design of Reliable Fixed-Point LTI Filters". In: *IEEE Transactions on Computers* 69.4 (2019), pp. 489–504. (cit. on pp. 685, 690, 694, 696).
- [VKG14] Martin Vetterli, Jelena Kovačević, and Vivek K. Goyal. *Foundations of signal processing*. Cambridge University Press, 2014. (cit. on pp. 670, 672, 674, 683, 684).
- [VLH17] Anastasia Volkova, Christoph Lauter, and Thibault Hilaire. "Reliable verification of digital implemented filters against frequency specifications". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2017, pp. 180–187. (cit. on p. 685).
- [Vol+19] Anastasia Volkova, Matei Istoan, Florent de Dinechin, and Thibault Hilaire. "Towards Hardware IIR Filters Computing Just Right: Direct Form I Case Study". In: *IEEE Transactions on Computers* 68.4 (2019). (cit. on p. 697).
- [VP07] Yevgen Voronenko and Markus Püschel. "Multiplierless Multiple Constant Multiplication". In: *ACM Transactions on Algorithms* 3.2 (2007). (cit. on p. 697).
- [WK08] Bernard Widrow and István Kollár. *Quantization Noise: Roundoff Error in Digital Computation, Signal Processing, Control, and Communications*. Cambridge University Press, 2008. (cit. on pp. 685, 689).
- [YLY17] Wen Bin Ye, Xin Lou, and Ya Jun Yu. "Design of Low Power Multiplierless Linear-Phase FIR Filters". In: *IEEE Access* (2017), pp. 23466–23472. (cit. on p. 697).



CHAPTER 24

Arithmetic for Deep Learning

Real stupidity beats artificial intelligence every time

Terry Pratchett, *Hogfather*

Machine learning has an ever increasing impact on many applications. This chapter studies the specific arithmetic requirements of deep neural networks (DNNs). The underlying number formats are considered in the context of DNNs and their training. DNN-specific arithmetic implementation issues are also discussed, with a focus on inference.

Machine learning is a hot topic that has already enabled many new applications, from intelligent personal assistants to self-driving cars, from natural language processing and translation to drug discovery, among many others. It has impacted most scientific fields, and many aspects of society at large. It is computationally very demanding, which makes it a good candidate for application-specific arithmetic.

We focus here on deep learning, a subtopic of artificial neural network (ANN)-based machine learning which is itself a subtopic of the artificial intelligence (AI) field. ANNs are inspired by the organization of the brain, where it is commonly accepted that computations take place in a densely connected network of neurons. Hence, an ANN is organized as a network of connected neurons (sometimes called a perceptron), usually organized in several layers (multi-layer perceptron). In a deep neural network (DNN), the number of layers ranges from five to more than a thousand [Sze+17]. ANNs date back to the 1940s, but they had to wait until the twenty-first century before Moore's law could provide enough computational power to train deep networks [GBC16], another key enabler being the huge amounts of training data provided by the internet era. The turning point in the rise

of deep learning was its success in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). It was won in 2012 by a DNN called AlexNet [KSH12], with a remarkable distance to its competitors. Since then, deep neural networks (DNNs) have consistently dominated this challenge, until beating human classification performance in 2015 [He+16].

These DNNs all belong to the important class of convolutional neural networks (CNNs), where the matrix multiplication used in generic DNNs is replaced by a computationally simpler convolution in most layers [GBC16]. Although their first success was in *image classification* problems, where the goal is to classify the content of an image (e.g., a picture showing a cat), DNNs have also been successful at many other problems. *Object detection* consists in identifying several objects in an image (i. e., providing for each object its classification but also its location in the image and its bounding box) [Red+16]. Apart from image processing, the field of *natural language processing* including *speech recognition*, *text to speech*, or *machine translation* has been heavily influenced by deep CNNs in the last decade.

DNNs are both compute- and memory-intensive [Sze+17; VKB18]. Application-specific arithmetic will help improve in both aspects: Computational limitations can be tackled by developing customized arithmetic tailored to the application. The memory footprint can be reduced by using data formats that are no larger than needed.

This chapter focuses on arithmetic issues in deep learning. Other important topics such as the computational architectures and their scheduling, or the theory and training of deep neural networks are only briefly introduced. A comprehensive introduction into deep learning can be found in textbooks [Agg18; GBC16]. An excellent survey about the implementation of deep neural networks can be found in [Sze+17]. Our reader should be aware that the field is moving very fast and information is quickly outdated.

This chapter first gives a general introduction to neural networks in Sect. 24.1 and then introduces the important class of CNNs in Sect. 24.2. Section 24.3 presents the diversity of number formats used in machine learning, and Sect. 24.4 discusses their effects on the training phase. Section 24.5 covers other implementation issues. As the best DNN architecture strongly depends on the application constraints, Sect. 24.6 presents a few case studies to show how the discussed techniques are applied in practice.

24.1 Neural Network Overview

24.1.1 Basic Neural Network Structure

The structure of a general neural network is illustrated in Fig. 24.1. It consists of several layers of data, which are called *activations*. The first layer is called the input layer and represents the input activations. The remaining layers

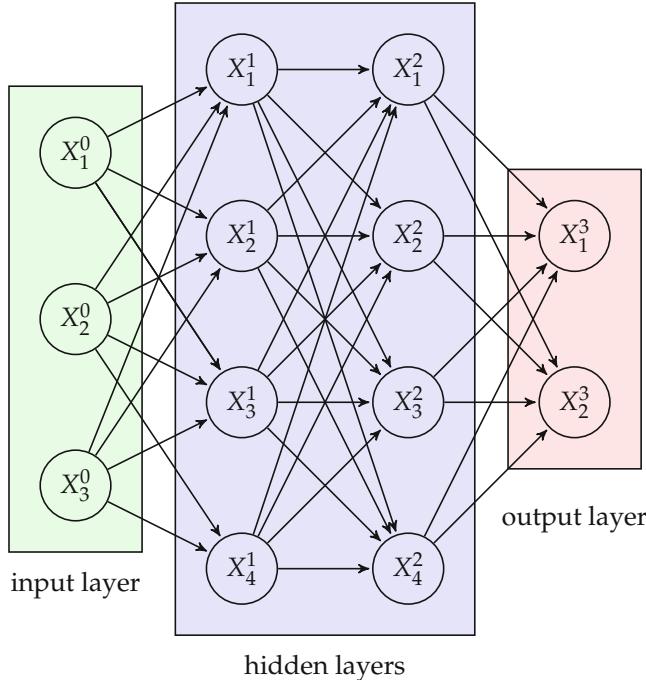


Fig. 24.1 Topology of an artificial neural network.

represent the outputs of artificial neurons. The last layer is called the *output layer*, and it holds the result of the computation. The intermediate layers are called the *hidden layers*.

Figure 24.1 also shows the data dependencies (which activation is computed out of which). If, like on this figure, data flows only from input to output without recursions, the network is called a feedforward network or multilayer perceptron (MLP); otherwise, it is a feedback network or recurrent neural network (RNN). Feedforward CNNs are typically used in image classification or object detection. Transformers [Wol+20] are another example of feedforward networks, often found in the processing of speech or other time-varying data. An example of a recurrent network is the long short-term memory (LSTM) network [HS97] which is as well used to process time-varying data.

An artificial neuron inputs activations X_j^ℓ and computes an activation $X_i^{\ell+1}$ as follows:

$$X_i^{\ell+1} = f \left(\underbrace{\sum_j W_{i,j} X_j^\ell + B_i^\ell}_{=Y_i^\ell} \right), \quad (24.1)$$

Table 24.1 Notations used in this chapter.

X_i^ℓ	Activation i of layer ℓ (may have several dimensions)
$W_{i,j}^\ell$	Weight used to compute output activation i in layer $\ell + 1$ from input activation j in layer ℓ
Y_i^ℓ	Intermediate sum of weighted input activations for output activation i in layer ℓ
B_i^ℓ	Bias i of layer ℓ
$f()$	Activation function
$F^*(\cdot)$	Global function that the network approximates
n^ℓ	number of activations in layer ℓ
W_i^ℓ	Weight i of convolutional kernel in layer ℓ (may have several dimensions)
$A_x^\ell, A_y^\ell, A_z^\ell$	Dimension of activation of layer ℓ in x, y and z direction
$K_x^\ell, K_y^\ell, K_z^\ell$	Dimension of kernel in layer ℓ in x, y and z direction
L_j	Loss value of output j
$\nabla_{i,j}^\ell$	Gradient of the loss with respect to $W_{i,j}^\ell$

where X_i^ℓ denotes the i -th activations at layer ℓ , $W_{i,j}^\ell$ is the weight contribution of input activation j to compute output activation i , B_i^ℓ is a *bias*, and $f()$ is an *activation function* (discussed in detail in Sect. 24.1.2).

The notations used throughout this chapter are summarized in Table 24.1. They also appear in Fig. 24.2, which illustrates the data dependencies from one layer to the next.

In Fig. 24.2, every output activation is computed from every input activation. This type of layer is called *fully connected* or *dense* layer. Non-dense layers are also possible and useful, in particular the convolution layers that will be introduced in Sect. 24.2.

As each layer is a vector of activations, it is natural to rewrite (24.1) using matrices. For this, we define the activation vector of layer ℓ

$$\mathbf{X}^\ell = \left(X_1^\ell \ X_2^\ell \ \dots \ X_{n^\ell}^\ell \right)^\top, \quad (24.2)$$

where n^ℓ denotes the number of activations in layer ℓ . The weights are organized in a single $n^{\ell+1} \times n^\ell$ matrix

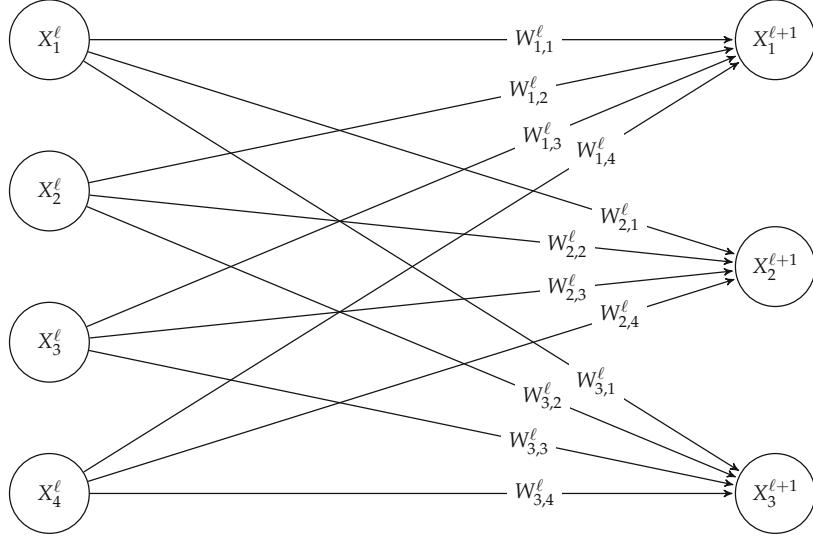


Fig. 24.2 Weights used between layers.

$$\mathbf{W}^\ell = \begin{pmatrix} W_{1,1}^\ell & W_{1,2}^\ell & \dots & W_{1,n^\ell}^\ell \\ W_{2,1}^\ell & W_{2,2}^\ell & \dots & W_{2,n^\ell}^\ell \\ \ddots & & & \\ W_{n^{\ell+1},1}^\ell & W_{n^{\ell+1},2}^\ell & \dots & W_{n^{\ell+1},n^\ell}^\ell \end{pmatrix}, \quad (24.3)$$

and the bias to

$$\mathbf{B}^\ell = (B_1^\ell \ B_2^\ell \ \dots \ B_{n^{\ell+1}}^\ell)^\top. \quad (24.4)$$

With that, (24.1) can be represented using matrix operations as

$$\mathbf{X}^{\ell+1} = \mathbf{f}(\mathbf{W}^\ell \mathbf{X}^\ell + \mathbf{B}^\ell), \quad (24.5)$$

where function \mathbf{f} is the vector version of f , applying f element-wise.

Note that activations often have a higher dimension. For instance, image processing often involves multidimensional layers with width, height, as well as additional *channel* dimensions. Colored images in the input layer typically have three channels for Red, Green, Blue (RGB). In hidden layers, there may be many more channels, corresponding to some rather abstract *features* detected by the network. For this reason, the activations in a specific layer are often called *feature maps*. With several channels, the computation can be described using tensors in a straightforward generalization of (24.5).

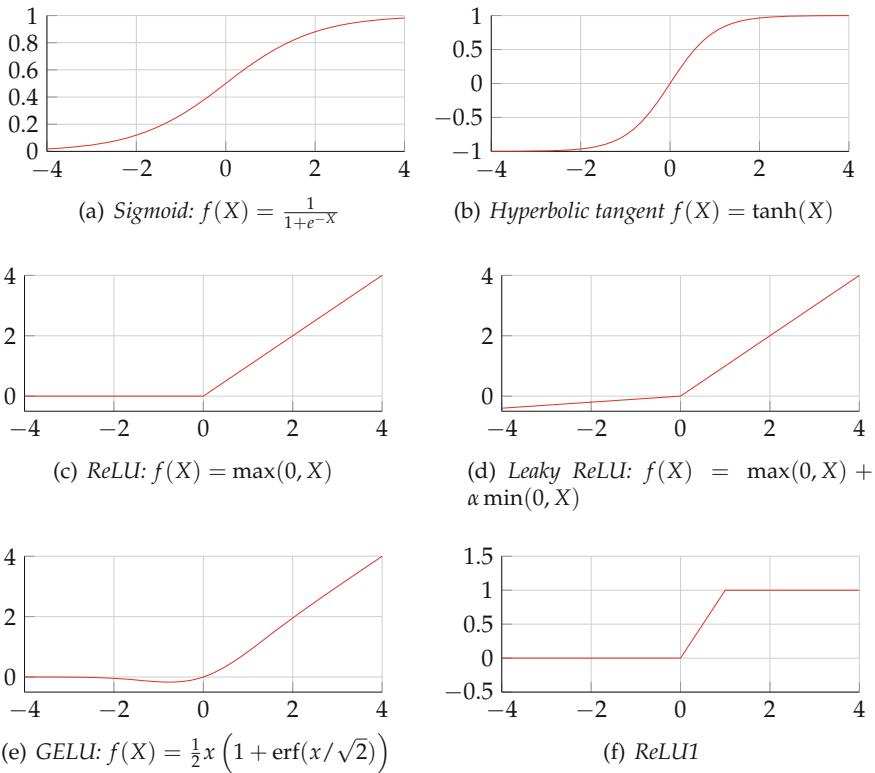


Fig. 24.3 Common activation functions.

24.1.2 Activation Functions

The unary function f in (24.1) is called the activation function. Without activation functions, a single layer of the previous neurons can compute any linear transformation, and adding more layers does not improve this expressivity. The main point of the activation function f is to improve the expressive power of neural networks. For this, f has to be nonlinear.

Earlier works have used functions such as the logistic sigmoid function, shown in Fig. 24.3a, and the hyperbolic tangent, shown in Fig. 24.3b. Those are approximately linear around zero and saturate for large positive or negative values.

Modern neural networks often use the rectified linear unit (ReLU) function [Hah+00], shown in Fig. 24.3c. It is much simpler to evaluate (both in software or hardware) and performs equally good or even better than previous nonlinear functions [NH10] despite having a single point of nonlinearity. It is even regarded as the default recommendation to start with [GBC16].

Several variations on the ReLU function have been introduced. Some involve a nonzero slope α [GBC16] for negative inputs, as illustrated by Fig. 24.3d. For $\alpha = -1$, it results in $f(X) = |X|$ (*absolute value rectification*). When α is fixed to a small value like 0.01, it is called *leaky ReLU*. The utility of these variants will be discussed when we study the training of neural networks in Sect. 24.4.

Another extension is the Gaussian error linear unit (GELU) [HG16], a smooth version of ReLU (see Fig. 24.3e) that improves accuracy but is much more costly to evaluate due to the use of the Gaussian error function (erf).

The ReLU1 function shown in Fig. 24.3f (a piecewise linear version of sigmoid or tanh) is interesting for a different reason: it ensures a priori that the output activation range is $[0, 1]$, thus simplifying the choice of its number format.

A general advantage of piecewise-linear activation functions (ReLU, ReLU1 and leaky ReLU) is that they are nonlinear but still offer some of the benefits of linearity. For instance, it is possible to apply some linear scaling to the weights or activations of a trained network, e.g., to normalize them in order to optimize data formats [CDP22], and compensate that in the activation function so that the network functionality is unchanged.

Note that there can be different activation functions in different layers, but this is uncommon in practice.

24.1.3 Topology and Hyperparameters

The overall network structure, also called *topology* of the network, describes the number of layers, their sizes, their type (fully connected, convolutional, etc.) and the activation functions to use for each layer. The parameters of the topology are called *hyperparameters*, to distinguish them from the parameters of the network itself, which are its weights and biases. The parameters are learnt during training (see next section), while the hyperparameters are fixed a priori.

A good network topology for a given problem is usually obtained empirically, based on experience with similar problems and also on the computing power available [LeC19]. Determining good topologies automatically is still a tedious and time-consuming problem [TL19] sometimes called hyperparameter optimization [FH19]. We do not discuss it further as it is mostly out of scope of this book.

24.1.4 Training and Using a Neural Network

The goal of the network is to approximate some global function

$$\mathbf{X}^{\ell_{\text{out}}} \approx F^*(\mathbf{X}^0), \quad (24.6)$$

where ℓ_{out} denotes the output layer. This is performed by adjusting the weights and biases for the whole network in a process called *learning* or *training*. The network topology is chosen before the training and then does not change. Training mostly concerns weights and biases and a few other parameters that will be introduced soon.

Once the network is trained for a given approximation, the network can be used on actual input data, which is commonly called *inference*.

Note that in contrast to the function approximation methods discussed in Part III (which essentially address functions of one variable), we are talking here about functions F^* of a very large number of variables. For instance, each pixel of an input image is a variable (or even three variables when coded in RGB), so the input to an image classifier neural network is a vector with many dimensions. Output data also has many dimensions (e.g., classes of a classifier). In addition to that, only relatively few values of the function F^* to approximate are actually known (e.g., by manual classifications in a classifier). Again, this is in contrast with Part III where the functions to implement were well-defined mathematical objects.

The purpose of learning is to create a network that generalizes the function, not one that only works for the data used for training (an issue known as *overfitting*). The definition of good datasets for training is a problem per se, and the AI communities use well-accepted benchmark datasets (such as MNIST, CIFAR, or ImageNet just for image classifiers) on which neural networks can compete.

The standard way to train a feedforward network is by using *backpropagation* [RHW86]. The training will be detailed in Sect. 24.4 but a short overview is given in the following. Each training iteration beginning with an inference pass on some training data using the current weights. A *loss function* measures the error between the obtained output and the desired value $F^*(\mathbf{X}^0)$ as a real number. Backpropagation uses the chain rule to determine the gradient of each weight with respect to this loss function from the output to the input. This gradient is used to change each weight by a small amount toward a smaller loss. Different algorithms exist how to apply this for many data points. A prominent example is the Stochastic gradient descent (SGD) algorithm (see Sect. 24.4 below). What is important at this point is that training is much more computation-intensive than inference, since training consists of many elementary training steps, each of which begins with an inference.

In *supervised learning*, the network is trained using examples that have been selected and prepared by humans. It is therefore also expensive in human labour. However, once training is done, the trained network can be used for arbitrarily many inferences.

Research is also active in *unsupervised learning*, for instance, reinforcement learning, where the networks learn from their environment by trial and error. This chapter focuses on supervised learning where most of the

arithmetic studies have taken place so far, but these studies are expected to transfer to unsupervised learning as this field matures.

24.1.5 Training Metrics

The loss function introduced above estimates the quality of a single feed-forward evaluation. To judge about the general performance of a network, several application-dependent metrics exist. They are usually evaluated on data which was not used in the training.

For classifiers, a common metric is the *accuracy* of a network, which is defined as the number of correct classifications divided by the total number of cases in the test set. However, classifiers typically compute a numerical score for each class. Considering this, top- n accuracy is defined as the percentage of inputs whose correct classification belongs to the n classification results with highest score. Often the top-1 and top-5 accuracies are reported when evaluating classifier networks. Here, the top-1 accuracy is most important as it states how often the network is correct. Per definition, the top- n accuracy is always lower or equal to the top-($n + k$) accuracy (for $k > 0$).

For object detection, which classifies several objects and their bounding box in an image, two metrics are common. The Intersection over Union (IoU) considers the intersection of the predicted bounding box in relation to the ground truth bounding box. The mean Average Precision (mAP) incorporates both classification accuracy and bounding box prediction.

Of course, other machine learning applications bring other metrics, but the idea remains the same: a local loss function is used to estimate the quality of a single evaluation, while a global metric is used to evaluate the performance of the whole network.

24.2 Convolutional Neural Networks

CNNs are a specialized kind of neural network for processing data that has a grid-like topology. An example of grid-like data is a time series of data (like an audio or radio signal): it can be seen as a one dimensional grid. Another example is an image, where the pixels are organized in a two-dimensional grid.

For such grid-like data, it is beneficial to replace, in some of the layers, the general matrix multiplications as given in (24.5) by the much cheaper *convolution* operation, described below. These layers are then called *convolutional layers*. A CNN typically consists of convolutional layers in the first layers and fully connected layers in the last layers (often only 1–2 layers remain fully connected).

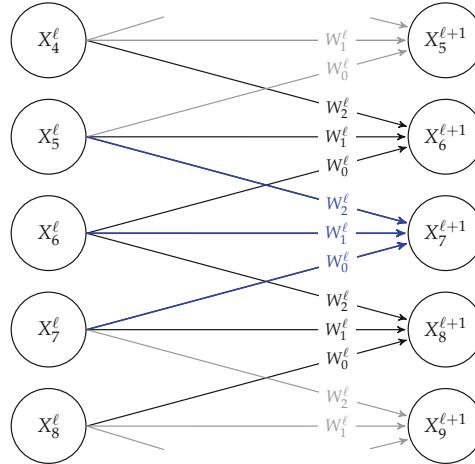


Fig. 24.4 Computation in a one-dimensional convolution with three weights.

A CNN is a neural network that includes at least one convolutional layer, but CNNs usually also involve other types of layers that we review now.

24.2.1 Convolutional Layers

In a convolutional layer, the generic matrix multiplication (the term Y_i^{ℓ} in (24.1)) is replaced by a convolution.

A one-dimension convolution

is defined as

$$Y_i^{\ell} = \sum_k W_k^{\ell} X_{i-k}^{\ell} \quad (24.7)$$

and illustrated by Fig. 24.4. Instead of a matrix of weights, we have a smaller (one-dimensional) vector

$$\mathbf{W}^{\ell} = (W_0^{\ell} \ W_1^{\ell} \ W_2^{\ell} \ \dots \ W_{n^{\ell+1}}^{\ell})^{\top}, \quad (24.8)$$

with $n^{\ell+1}$ denoting the number of weights in layer $\ell + 1$. The convolution can be written in short by using the *convolution operator* (also see Sect. 23.1.4, p. 672)

$$\mathbf{Y}^\ell = \mathbf{W}^\ell \circledast \mathbf{X}^\ell. \quad (24.9)$$

The weights are often referred to as the *kernel* or *filter* of the layer, and this kernel is applied to each data point of the grid. One kernel is highlighted in Fig. 24.4. The output activation is then computed using a bias and an activation function like in the generic ANN:

$$\mathbf{X}^{\ell+1} = f(\mathbf{Y}^\ell + \mathbf{B}^\ell). \quad (24.10)$$

Using the matrix representation introduced above, the convolutional layer can be seen as a special case of the fully connected layer, using in (24.3) a matrix with a specific structure. For the example of a kernel with three weights, the matrix has the form

$$\mathbf{W}^\ell = \begin{pmatrix} W_2^\ell & W_1^\ell & W_0^\ell & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & W_2^\ell & W_1^\ell & W_0^\ell & 0 & \dots & 0 & 0 & 0 & 0 \\ & & & & & & \ddots & & & \\ 0 & 0 & 0 & 0 & 0 & \dots & W_2^\ell & W_1^\ell & W_0^\ell & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots & W_2^\ell & W_1^\ell & W_0^\ell \end{pmatrix}, \quad (24.11)$$

which obviously is sparse compared to (24.3). Besides, it contains a lot of redundancy. Compared to the fully connected layer (see Fig. 24.2), each output activation depends on much fewer input activations, and re-uses the same weights. Hence, a CNN layer requires much lower computational effort and a much lower memory footprint for storing the weights.

In the two-dimensional case

the discrete convolution is defined as

$$Y_{x,y}^{\ell+1} = \sum_m \sum_n W_{m,n}^\ell X_{x-m, y-n}^\ell. \quad (24.12)$$

Here the indices x and y typically belong to a symmetrical set $\{-k, \dots, 0, \dots, k\}$. Figure 24.5 illustrates the convolution for a 3×3 kernel applied on a single channel.

Typical two-dimensional CNNs process several channels. For example, as illustrated in Fig. 24.6, the input layer typically receives an image of size $A_x^\ell \times A_y^\ell$ with $A_z^\ell = 3$ RGB color channels. Subsequent layers then tend to reduce A_x and A_y while augmenting A_z (remember that a channel is a “feature” detected by the network).

Each activation of layer $\ell + 1$ is computed out of the weighted sum of the activations within the $K_x^\ell \times K_y^\ell \times K_z^\ell$ kernel represented by the green rectangular box in Fig. 24.6. However, the green block does not represent

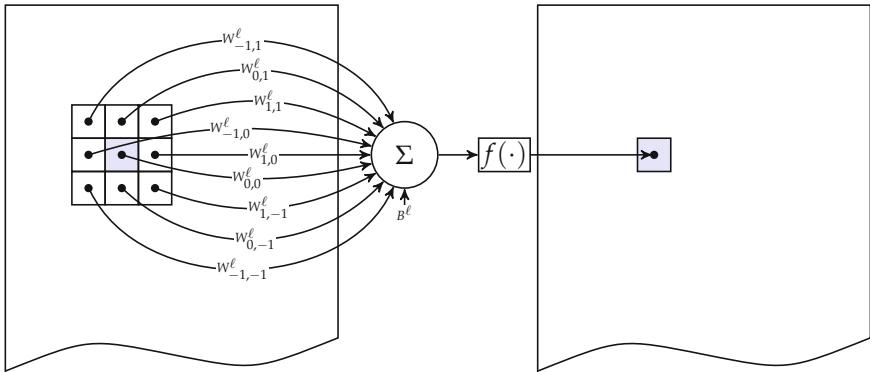


Fig. 24.5 Illustration of a two-dimensional convolution layer.

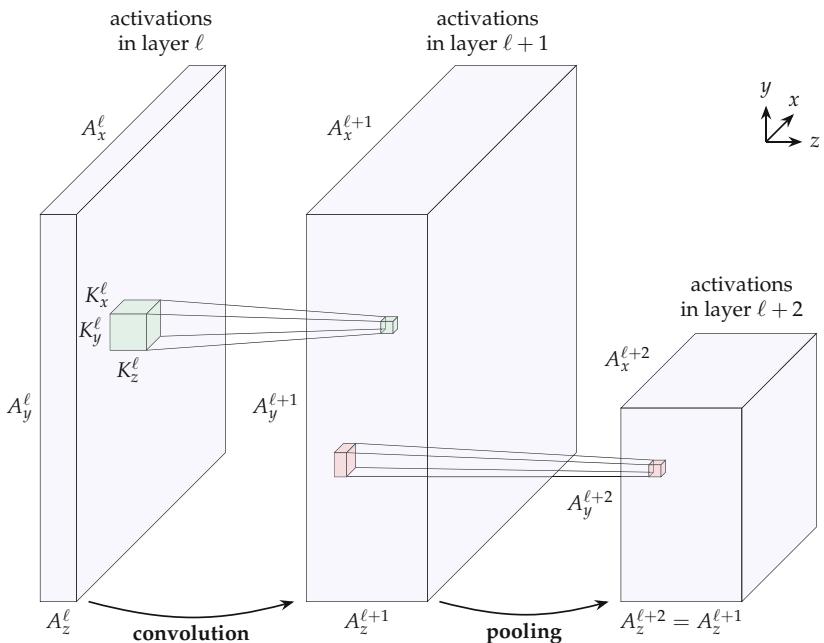


Fig. 24.6 Illustration of the multichannel convolution and pooling operations.

a three-dimensional convolution. Instead, a two-dimensional convolution is performed for each channel, and the sum of these convolutions gives the output value as given by

$$Y_{x,y,z}^{\ell+1} = \sum_{z'} \sum_m \sum_n W_{m,n,z'}^{\ell,z} X_{x-m,y-n,z'}^\ell. \quad (24.13)$$

The term $W_{m,n,z'}^{\ell,z}$ denotes the weights of the $K_x^\ell \times K_y^\ell \times K_z^\ell$ kernel that is used to compute output channel z in layer ℓ .

The kernel depth K_z^ℓ is equal to the activation depth (or channel size) A_z^ℓ , i.e., $K_z^\ell = A_z^\ell$. Each of the $A_z^{\ell+1}$ output channels is processed by an individual kernel. Thus, $A_z^{\ell+1}$ is equal to the number of kernels needed in the computation of layer $\ell + 1$. Altogether, the number of different scalar weight values in the convolutional layer ℓ is $K_x^\ell \times K_y^\ell \times K_z^\ell \times A_z^{\ell+1}$.

Practical Considerations

It is often desirable that the width and height are not changed by the convolution ($A_x^{\ell+1} = A_x^\ell$ and $A_y^{\ell+1} = A_y^\ell$), either for implementation considerations such as memory alignment or to match the requirements of the application. To achieve this, the inputs have to be padded at the border by either zeros or another constant (*constant padding*) or by replicating data in the image (*reflection padding* or *replication padding*). If no padding is performed, the output size will slightly shrink.

It is also possible to reduce the output size by shifting the kernel not by one but by a positive value called *stride*. A stride of two in one dimension approximately halves the output. However, strides in convolutional layers are more rarely used than *pooling layers*, which are introduced next.

24.2.2 Pooling Layers

CNNs often use an additional layer type called *pooling layer*, typically placed after a convolutional layer. As illustrated by the right part of Fig. 24.7, a pooling layer combines neighboring activations into a single activation output. This has several advantages [BPL10]: it reduces the amount of data, it makes the network more robust to noise, and also more robust to small translations in the input data. Pooling is therefore a key component of several state-of-the-art CNNs.

In a pooling layer, the data can be combined in various ways [GBC16]. The *max pooling* operation selects the maximum out of neighboring activations. Figure 24.7a shows an example of max pooling on a two-dimensional image using a 2×2 window.

Average pooling computes the average of all the neighboring activations. In *weighted average pooling*, the activations are weighted with respect to their distance to the center. Figure 24.7b shows an example of average pooling on a two-dimensional image using a 2×2 window.

While average pooling was used in early works, most modern CNNs use max pooling. However, the best pooling method seems to depend on the problem [BPL10].

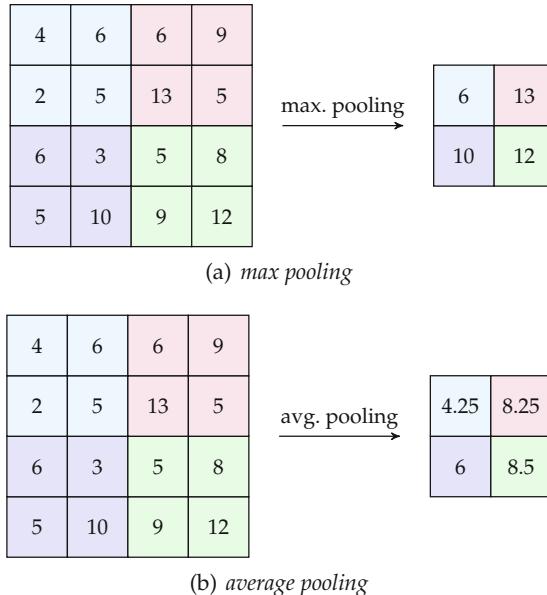


Fig. 24.7 Common pooling methods.

On CNNs having multiple channels, the pooling is applied on a per channel basis, i.e., the number of channels does not change. This is illustrated on the right of Fig. 24.6 showing the pooling by the red box of size $2 \times 2 \times 1$ from layer $\ell + 1$ to a $1 \times 1 \times 1$ cube in layer $\ell + 2$. To effectively reduce the data, a stride is used of the size of the pooling window. Hence, in our example, the window is moved by two positions (which corresponds to a stride of two). This effectively halves the width and height (i.e., $A_{x/y}^{\ell+2} = A_{x/y}^{\ell+1}/2$ in Fig. 24.6) but does not change the channel size (i.e., $A_z^{\ell+2} = A_z^{\ell+1}$).

24.2.3 Batch Normalization

Performing inference tasks on a batch of images instead of a single image is more efficient, as it reduces the memory bandwidth requirements (the weights of a layer are reused for all the images in a batch and are therefore kept in cache for a longer time). Besides, batch processing opens another opportunity: assuming that all the images in a batch are statistically independent, it is possible to attempt to normalize the means and variance of each activation in a batch. This is done by first subtracting from each activation X its mean μ_B over the batch B , then dividing by the square root of its variance σ_B^2 to get

$$\hat{X} = \frac{X - \mu_B}{\sqrt{\sigma_B^2}}, \quad (24.14)$$

which has zero mean and unit variance. Finally, each activation is normalized as

$$X_{\text{norm}} = \gamma \hat{X} + \beta, \quad (24.15)$$

where γ and β are two parameters that have to be learned for each dimension of the activation. This greatly reduces training time while improving accuracy [IS15].

The parameters μ_B and σ_B depend on the values processed by the batch and evolve during training. Conversely, γ and β must be learned. Therefore, in the training phase, batch normalization needs to appear as a specific layer so that it is integrated in the backpropagation algorithm.

For inference, μ_B and σ_B can be replaced with the mean and variance over the whole training set. Then, the batch normalization corresponds to a linear transform that can be merged in the weights of existing layers [IS15]: there is no need for batch normalization layers in inference architectures, even if they still process data in batch for performance reasons.

24.2.4 Passthrough Layer

Passthrough layers were introduced concurrently for Darknet in the YOLO detector [Red+16; RF16] and in a similar form in ResNet [He+16]. They are also called *shortcut connection* and the networks containing them *residual networks*. The idea is to combine activations from several preceding layers without further processing. The activations of layer ℓ no longer only depend on layer $\ell - 1$ but also on other previous layers ($\ell - 2, \ell - 3, \dots$). As the resolution is shrinking with every pooling layer, this allows to still use information from higher-resolution layers in later layers.

24.2.5 Softmax Layer

This post-processing layer is typically found at the output of classifiers. It normalizes a vector of activations into a vector that sums up to 1. Each activation can then be interpreted as the probability of belonging to a certain class. Note that this is just an interpretation without any real statistical significance. However, it is convenient, compared to a set of activations that can take any value, including nonpositive ones.

Mathematically, this layer computes the softmax function defined as

$$\sigma(X_i) = \frac{e^{X_i}}{\sum_j e^{X_j}}, \quad (24.16)$$

which has the property that

$$\sum_i \sigma(X_i) = 1 \quad (24.17)$$

for any real vector $X_i \in \mathbf{X}$. Computing the exponentials and the division is quite expensive, and there have been several attempts to approximate the softmax function [Shi+17; KP20].

The softmax function can be seen as a smoother version of the arg max function: it is continuous and differentiable, which is important for training [GBC16]. However, for inference, in the common case when only the single output with maximum probability is required (classifiers), the softmax can be replaced by the much cheaper arg max function that will provide a one-hot representation of the result.

24.2.6 An Example CNN

Figure 24.8 shows a typical classifier CNN, consisting of several convolutional and pooling layers followed by two dense (fully connected) layers, and a softmax layer [SZ15; Tri+19]. It is a variant of the family commonly denoted VGG,¹ which was also used in [LZL16; Tri+19] (named VGG-7). It utilizes convolutional layers with small (3×3) kernels and only ReLU activation functions [SZ15]. The input is an image of 32×32 pixels having three colors each. With a final layer consisting of ten neurons, it implements a classifier that can distinguish between ten classes (e.g., for the CIFAR-10 dataset²).

24.3 Number Formats

Better number formats for machine learning are still being actively researched at the time of writing this book, and the present section may well be obsolete a few years after publication.

The general trend is toward smaller and simpler formats, with the main goal of reducing memory footprint and memory traffic. Memory footprint

¹ Visual Geometry Group (VGG) is the name of the group at Oxford who showed that, compared to the previous state of the art, accuracy could be improved by more convolution layers with smaller kernels [SZ15].

² <https://www.cs.toronto.edu/~kriz/cifar.html>

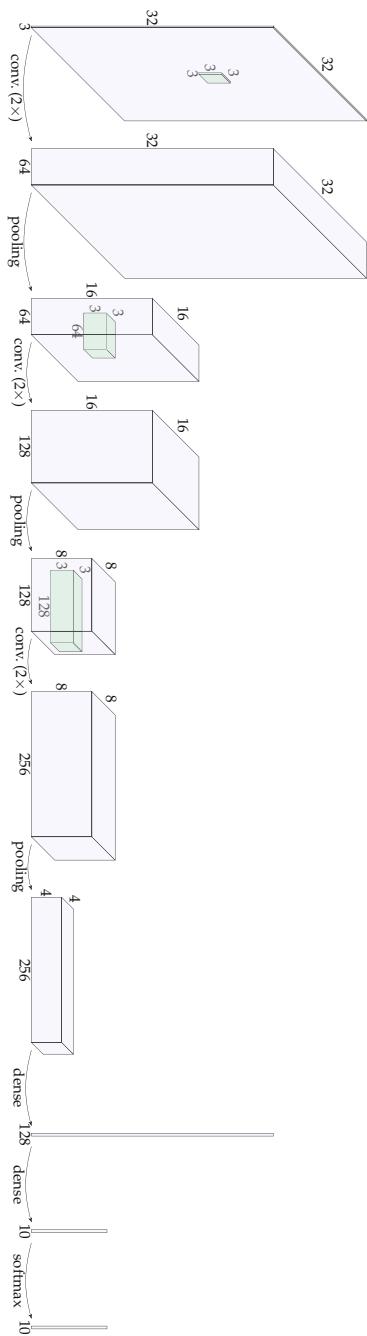


Fig. 24.8 An example of image classification CNN.

describes the total size of network parameters (the weights, the biases, etc.). Memory traffic also includes the activations which are buffered between layers, and, during training, the gradients and other parameters. Memory traffic is of crucial importance in terms of performance and energy consumption. It is definitely improved by reducing the size of the data, which is an arithmetic problem. On top of this, the study of memory access patterns is another active research topic, but it is out of the scope of the present book.

New machine learning algorithms are initially developed using binary32 floating-point arithmetic, with all the weights, biases and activations stored as binary32 numbers. This arithmetic is available in most processors and graphics processing units (GPUs), and the wider binary64 format brings no noticeable improvement. The move to smaller formats is an optimization that is considered when this initial development is satisfactory. When measuring the application-level accuracy achieved using smaller formats, the reference is almost always the binary32 baseline.

In terms of number formats, a distinction should be made between inference (addressed in Sect. 24.3.1) and training (addressed in Sect. 24.3.2).

The main difference is that training requires a larger dynamic range than inference. Actually, many improvements in training techniques have consisted in reducing the dynamic range of the weights (it is one of the roles of batch normalization for example), but the range required in the early training iterations is difficult to predict. Furthermore, this range evolves during the training process. For this reason, training is typically performed in floating point or Logarithm Number System (LNS).

Conversely, most inference tasks can be performed in fixed point with very small formats (8 bits or less), provided the training is directed accordingly (quantization-aware training, see Sect. 24.4.4).

Another difference is that training is often performed on hardware that is more powerful than the inference hardware. For instance, a CNN eventually deployed for image recognition tasks on mobile phones has usually been trained in a data center.³ This is partly due to the need to access terabytes of training data and partly due to the fact that training is much more compute-intensive than inference. In such cases, training is less constrained in resources than inference.

It is also important to distinguish between the formats of the data, and the formats internally used for the computations, which is usually wider. This is true for training and for inference.

24.3.1 Number Formats for Inference

Floating-point formats for inference will be discussed in Sect. 24.3.2, and this section focuses on simpler formats and their associated arithmetic units, il-

³ There is also a lot of inference performed in data centers, in particular those related to search engines and social networks [LeC19].

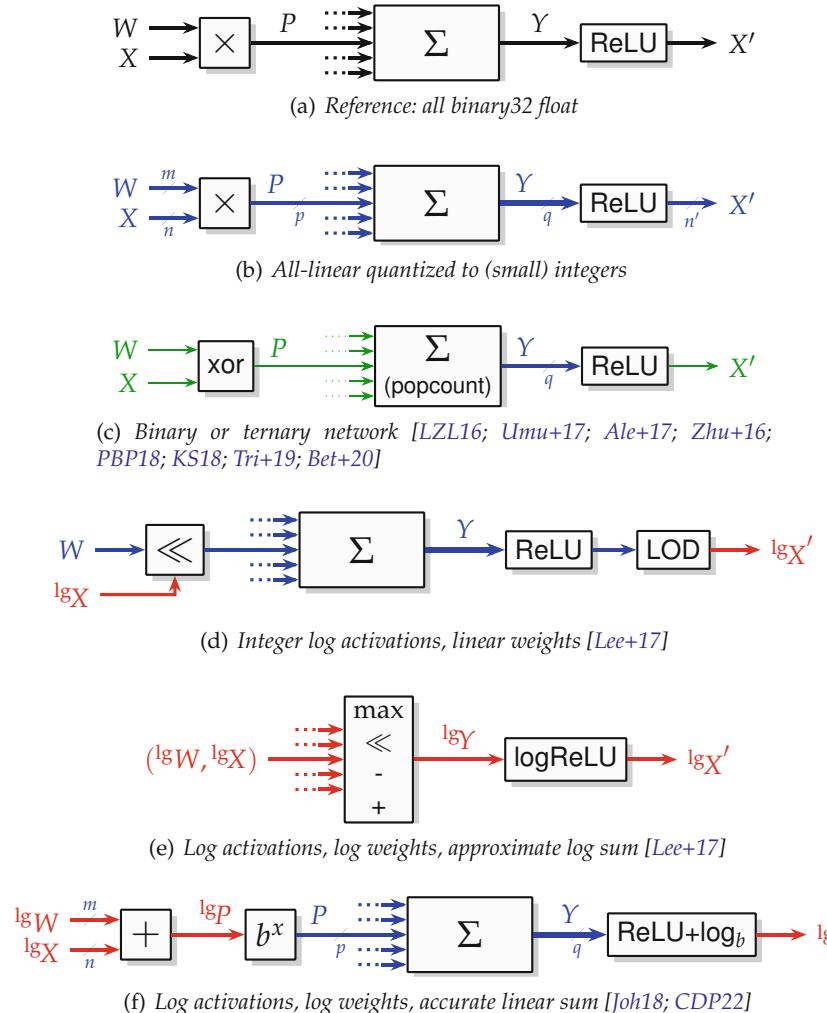


Fig. 24.9 Some arithmetic formats that have been proposed for artificial neurons: integer/fixed-point data in blue, binary/ternary data in green, log-encoded data in red.

Illustrated in Fig. 24.9. The weights, biases, and activations may use the same format, or not, as discussed below.

24.3.1.1 Fixed-Point Formats for Inference

Let us first assume post-training quantization (PTQ), which means that quantization of the weights is applied to the network after a complete training using binary32 floating-point format. Experiments have shown that all

the weights of a network can be quantized to the same 16-bit fixed-point format without degrading the accuracy [Gys+18; Jun+21]. This corresponds to $m = n = p = n' = 16$ on Fig. 24.9b. The scaling factors, i.e., the positions of the fixed point, are irrelevant here as all the computations before the ReLU are linear, and the ReLU itself is piecewise linear: therefore fixed-point formats can trivially be reduced to integers.

Choosing different custom formats for each of the layer inputs, layer weights, and layer outputs,⁴ it is possible to quantize weights and activations to 8 bits with only a slight degradation in classification accuracy [Gys+18].

Alternatively, quantization may be integrated into the training of the model: this is called quantization-aware training (QAT) and is discussed in Sect. 24.4.4. Then even lower precisions can be used: encoding the weights on 5 to 8 bits is typically sufficient to reach classification accuracies comparable with a binary32 floating-point network [Gys16; Jac+18].

A general rule here is that inference hardware uses more bits to compute the intermediate sum Y . In Fig. 24.9b, typically $p = m + n$ so the products are computed exactly, and $q \geq p$ so that the summation can be exact as long as no overflow occurs. As the value of Y does not need to be stored, this extra precision does not entail memory traffic. Besides, an accurate computation of the sums of products means that the hardware exhibits a behavior comparable to a binary32 simulation (inputting low-precision weights and biases, computing in binary32, and complemented with quantization layers behind the activation functions).

For illustration, with 8-bit weights and biases, the exact products fit on 16 bits, and accumulating them in a 24-bit Y will match the accuracy of a binary32 simulation with the same 8-bit data as long as no overflow occurs.

Since there is one ReLU for many additions and multiplication, its contribution to the cost is very small even if it inputs this wider sum format. The quantization to the output format (of activations) is also easy to integrate in the ReLU operation.

24.3.1.2 Ternary and Binary Formats

DNNs can even be trained down to ternary (weights and activations can only take the values $-1, 0$ or $+1$, scaled by a factor s) [Ale+17; And+18; PBP18] or even binary (scaled -1 or $+1$) [Ras+16; Umu+17; KS18; And+18; Bet+20]. This is illustrated in Fig. 24.9c. A variant is to train the weights to binary or ternary but to keep a wider precision for the activations [CBD15; Tri+19] (e.g., 4 bits in [LZL16; Zhu+16]). This still removes the need for multipliers.

⁴ The authors of [Gys+18] call this a *dynamic* fixed-point format, also named block floating-point in other contexts [Alt05]. However, the format does only depend on the layer and is not adjusted dynamically from a hardware perspective.

For such extreme quantizations, some form of QAT is required (see [Cou+16; Bet+20; Qin+20] for binary and [LZL16; Ale+17; Zhu+16] for ternary) and for a given network structure the classification accuracy will be degraded by a few percentage points compared to the binary32 reference network, ternary networks behaving slightly better than binary ones. This accuracy drop can be compensated by adding more channels or layers (it was shown theoretically that binary nets can implement any digital neural network [Con20]), but the corresponding overhead must not offset the benefit of using a reduced precision. At the time of writing this book, it remains an art to design a binary or ternary network.

Two such networks are presented as case studies in the sequel: the binary FINN framework [Umu+17] in Sect. 24.6.2 and a ternary network [Tri+19] in Sect. 24.6.4.

24.3.1.3 Logarithmic Formats

Finally, several works have considered for neural networks the use of Logarithm Number System (LNS) introduced in Sect. 2.6. The potential of low-precision logarithmic encoding was demonstrated [MLM16] then put in practice [Lee+17; Tan+17] with an architecture (Fig. 24.9d) that keeps the activations in the linear domain and implements multipliers as bit shifts. The same authors [Lee+17] also considered an architecture that performs an approximate sum in the logarithmic domain (Fig. 24.9e). Another approach is to replace, in an architecture using linear encoding (Fig. 24.9b), the standard multiplier with an approximate one using LNS arithmetic [Kim+18; ACJ20].

A fourth approach, illustrated by Fig. 24.9f, is the “somewhat inaccurately named” [Joh18] Exact Log-Linear Multiply-Add (ELMA) approach [Joh18; CDP22]. The term “exact” emphasizes the fact that both the addition of the logarithms and the summation of the product terms are exact, since both are performed as fixed-point additions. However, there are still rounding errors in the log/linear conversions (in the b^X and $\text{ReLU}+\log_b$ boxes of Fig. 24.9f). Since the summation is performed in fixed point, as above, it is a good idea to have $q \geq p$. Finally, most studies use base $b = 2$, but b is actually a parameter that can be used to fine-tune the dynamic range of a format for a given number of bits [AGG21].

Several tricks can be used to save bits in the LNS representation [CDP22]. They are illustrated in Fig. 24.10, a more detailed version of Fig. 24.9f. One is the systematic use of ReLU1 (see Fig. 24.3, p. 712) which ensures that activations are in $[0, 1]$. As $X \geq 0$, it does not need a sign bit; as $X \leq 1$, its logarithm is always negative and does not need a sign bit either. A nonzero activation X is then simply represented by $L_X = \lfloor -\log_b |X| \rfloor$, rounded to a fixed point number.

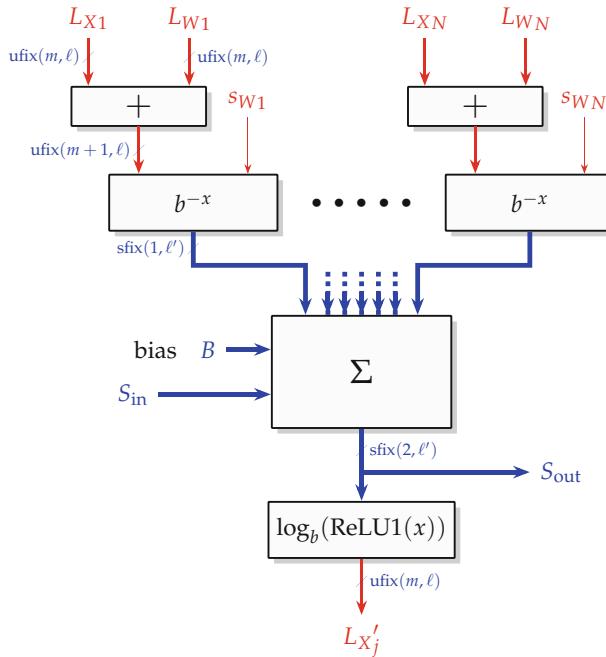


Fig. 24.10 An LNS low-precision neuron. [CDP22]

Weights are typically small in practice, so ensuring $|W| < 1$ is not an issue. However, weights can be negative or positive, so a weight W is represented by a sign bit s_W and the fixed-point value of $L_W = \lfloor -\log_b |W| \rfloor$.

A problem is that $\log_b(0)$ is undefined, so the previous representation cannot represent zero weights nor activations. Zero usually needs a special encoding in LNS. In the context of Fig. 24.10, however, a further trick is to use the largest representable value of L_X to represent 0, with a choice of formats such that the antilog conversion (the b^{-x} box) will convert this value to a true zero for the summation [CDP22]. Note that all the larger values, which correspond to multiplications by such a zero, are then also rounded to zero.

Altogether, this approach provides good accuracy for word sizes between 4 and 6 bit [CDP22]. For such very small formats, the log and antilog of Fig. 24.10 operations can be tabulated, and these tables are efficiently implemented using the look-up tables (LUTs) of field programmable gate arrays (FPGAs) (see Fig. 16.11, p. 492).

24.3.2 Number Formats for Training

The binary32 format (exponent range $[-126, 127]$, 23 bits of significand) is an overkill for training tasks. This has led to the definition of the nonstandard formats already introduced in Sect. 2.5.2, p. 52. Figure 24.11 shows the floating-point formats supported by mainstream hardware vendors at the time of writing this book.

The bfloat16 [HTH19; Bur+19; NVI20], DLFloat [Agr+19] and TensorFloat32 [NVI20] formats offer a dynamic range comparable to binary32 (8, 7, and 8 exponent bits, respectively) but much less precision (7, 8, and 10 significand bits, respectively, corresponding to between two and three decimal digits). It is possible to find hardware that supports them, and training with them is essentially as good as training with binary32. At the time of writing this book, the challenge is to use even smaller formats.

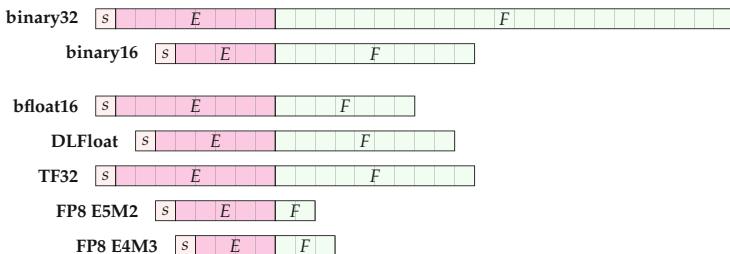


Fig. 24.11 Some floating-point formats used in machine learning.

There is a consensus in the industry [Sun+19; Tam+19; Mic+22; Nou+22] that 8-bit floating point (FP8) can be used in training tasks, including very large ones [Mic+22], with a small quality degradation compared to pure binary32 training. However, two different FP8 encodings are needed: E5M2 (an IEEEfloat(5,2) that can be viewed as the upper half of binary16) and E4M3 (a modified IEEEfloat(4,3) which drops NaN payloads and infinities in exchange for more normal numbers). The recommended use [Mic+22] is E4M3 for the weights and activations and E5M2 for the gradients computed by the training process. Besides, some care must be taken to the scaling of data (either using bias adaptation [Sun+19; Tam+19; Nou+22] or per-tensor scaling [Mic+22]).

Assuming that E4M3 weights are a good enough representation of the weights, here is an intuitive explanations of the need of a different format for the gradients. The essence of a training iteration is to compute a small update to each E4M3 weight. This small update is itself the result of a large computation. Performing this computation using E5M2 enables terms of smaller magnitude than using E4M3. The fact that E5M2 has one signifi-

cand bit less than E4M3 is not an issue if the update is small with respect to the weight.

Again we stress that the computations themselves must be performed with extended precision for all this to work.

Even smaller formats are being investigated, for instance, a 4-bit format that keeps only the exponent, with only one implicit bit of significand [Chm+22]. There is little difference between such a format and LNS, which has also been studied as a training format [ACJ20; Zha+21].

24.4 Training

Before we get into quantization-aware training, we have to briefly introduce the used *training data* and the training algorithms which are based on *back-propagation*.

24.4.1 The Training Data

Training uses a set of input-output pairs $(\mathbf{X}^0, \mathbf{X}_{\text{exp}}^{\ell_{\text{out}}})$. Here $\mathbf{X}_{\text{exp}}^{\ell_{\text{out}}} = F^*(\mathbf{X}^0)$ is the expected output when the input is \mathbf{X}^0 . This set approximates F^* , the global function defined in (24.6), in the sense that it does not cover all the possible inputs. For example, in an image classification problem, each pair consists of an input image (e.g., a picture of a cat) and the classification result (e.g., the label “cat”). A large number of training data is required for modern DNNs. For instance, the data set used in the ImageNet image classification competition contains 1,000 classes, each class being represented by 1,000 images [Den+09]. The training data is often classified by humans, and this can be a major cost when building a new machine learning application.

This data is then usually divided into two disjoint sets: the *training set* and the *validation set*. Only the training set is used during the actual training. Then, an unbiased evaluation of this training only uses the validation set. Thus, it is possible to check that the network is able to generalize what it has learnt on the training set. If the results are much better on the training set than on the validation set, it means that the network memorized input/output pairs but is unable to generalize them. This is called *overfitting*. Avoiding overfitting often requires changes in the network topology (e.g., reduce the size), in the training process (e.g., stop earlier), or in the training data (e.g., add synthetic data to the training set, balance the number of data samples per output class in a classifier, or remove redundant data).

Usually, the training set is larger than the validation set, but this is very problem-specific. A good starting point is to take 80% of the data for training

and 20% for validation [GBC16]. Note that the validation set is sometimes used to tune the model's hyperparameters (like the number of kernels in each layer, the size of the kernels, etc.) which requires a third disjoint unbiased set, often called the *test set*. When no hyperparameters are tuned, validation set and test set are often used synonymously.

24.4.2 Training by Backpropagation

Nearly all training algorithms for feedforward neural networks like Stochastic gradient descent (SGD) rely on backpropagation [RHW86]. They work from output to inputs, adjusting the weights in order to reduce the output error of the network.

The training usually starts with a random initialization of the weights. The actual distribution of these random weights depends on the activation function used [Agg18]. The training then consists of a number of *epochs*. In its simplest form, each epoch is itself an iteration on the whole of the training set, performing an elementary training step on each input/output pair $(\mathbf{X}^0, \mathbf{X}_{\text{exp}}^{\ell_{\text{out}}})$. Each training step consists of a forward evaluation using the current weights and then a backpropagation step that updates the weights.

The forward evaluation is simply an inference: from the input \mathbf{X}^0 , it computes an output $\mathbf{X}^{\ell_{\text{out}}}$. The error of the network on \mathbf{X}^0 is defined with respect to the golden reference $\mathbf{X}_{\text{exp}}^{\ell_{\text{out}}}$ as a real-valued *loss function* L :

$$L(\mathbf{X}_{\text{exp}}^{\ell_{\text{out}}}, \mathbf{X}^{\ell_{\text{out}}}) \in \mathbb{R}. \quad (24.18)$$

Possible loss functions are the mean squared error (MSE), the mean absolute error (MAE), or the cross-entropy [GBC16]. The loss function to use strongly depends on the application (e.g., classification, detection, regression) which determines the type of the output layer (e.g., whether it represents a linear value or a binary decision obtained from a softmax layer).

In (24.18), the loss function L is written as a function of the output value, but it is also a function of the weights and biases of the network itself. For backpropagation to work, it must be differentiable with respect to each of these parameters. More precisely, the gradient of the loss value L_j of each output j has to be computed for each of the weights that contributes to this output (the same applies to biases and we will not detail it further):

$$\nabla_{i,j}^{\ell} = \frac{dL_j}{dW_{i,j}^{\ell}}. \quad (24.19)$$

A positive gradient shows that a slight decrease of $W_{i,j}^{\ell}$ would also decrease the loss and vice versa. The core idea of learning is therefore to update the

weights according to

$$W_{i,j}^{\ell+} = W_{i,j}^{\ell} - \eta \nabla_{i,j}^{\ell}, \quad (24.20)$$

where η is a small positive real parameter called the *learning rate*. As long as η is small enough and $\nabla_{i,j}^{\ell}$ is nonzero, this update entails a small decrease in the loss function for this training pair. This *gradient descent* method dates back to Cauchy in 1847.

The previous shows how to update the weights and biases of the output layer. To update the parameters of previous layers, we need to compute the corresponding gradients of the loss function. Here, the chain rule is used to express the gradients in layer $\ell - 1$ in terms of the gradients in layer ℓ . The actual formulation is too technical to be detailed here, but simple, since the convolutions are linear and the derivatives of the activation functions used are known (and, in the case of ReLU variants, simple, too).

Therefore, the gradients can be propagated backward from the output layer to the input layer, hence the name *backpropagation*. For more details, we refer the interested reader to textbooks [Agg18; GBC16].

So far we have presented backpropagation based on one input-output pair of the training set. However, we have many such pairs in the training set, and each pair will yield a different gradient, hence different, possibly contradicting, updates to weights and biases. This suggests several training strategies:

- **Batch gradient descent** considers all the pairs in the training set and defines the loss function as the mean of all the individual losses. It is slow but provides a smooth loss reduction.
- **Stochastic gradient descent (SGD)** updates the weights according to a single random pair of the training set. It is obviously faster, but since a step only considers one pair, it may actually degrade the average performance of the network over the other pairs.
- **Mini-batch gradient descent** is a mix of the previous two. It takes a randomly shuffled subset of the training set (called a “mini-batch”) and computes the mean of all the individual losses from this subset. A proper choice of the mini-batch ensures a good speed trade-off as well as consistent performance improvement. This is currently the de facto standard training strategy.

An epoch is complete once all samples of the training set are evaluated. It typically takes many (hundreds of) epochs to train a DNN.

24.4.3 Training and Activation Functions

It was mentioned that the ReLU activation function is cheap to evaluate. It turns out that it is also a good choice as far as training is concerned. Training deep networks using sigmoid or tanh exposes the *vanishing gradient problem* [CBB11]: the products of gradients in the backpropagation training are getting so small that the training of the first layers effectively stops. With a derivative that is either 0 or 1, the ReLU function prevents this.

However, ReLU's saturation to zero for negative inputs leads to another issue, the *dying neurons problem* [Agg18], where some neurons constantly return 0. This is the problem that leaky ReLU was designed to solve. Once training is done, its parameter α can be set to zero to simplify inference.

24.4.4 Quantization-aware training (QAT)

To allow the smallest possible word sizes for the weights, biases, and activations, it is necessary to already consider the effect of quantization during training and fine-tune the weights and biases based on it.

This topic is a highly active research field at the time of the writing of this book. An excellent survey of current research can be found in [Gho+21].

The main idea is to consider the quantization functions as an integral part of the neural network. In fact, as discussed above, a nonlinearity is required anyway. So, it seems natural to train the network including this quantization. The problem here is that the quantization function is a non-differentiable step function, with a zero gradient almost everywhere [Gho+21], making it unusable during backpropagation.

So, we have to find ways to approximate the quantization function.

A naive idea consists in quantizing the weights at every weight update when evaluating (24.20). With very low precisions, a problem arises: as the training converges, the weight gradients are getting smaller and smaller, until they are small enough that a weight update is smaller than a quantization step. Then the weight will no longer change: this will effectively stop the convergence. A simple way out is the straight-through estimator (STE) approach [BLC13]. The idea is to keep the weights at full precision for the backward path but to evaluate the forward path using the quantized weights. Thus, even small gradients will cause small changes in the full precision weights that will continue converging and may eventually lead to different (hopefully better) quantizations after several iterations. By doing so, the non-differentiable quantization function is only applied in the forward path. When training is done, only the quantized weights are used for inference.

The state of the art in QAT is to perform regular floating-point training until a certain accuracy is reached. Then, any batch normalization parame-

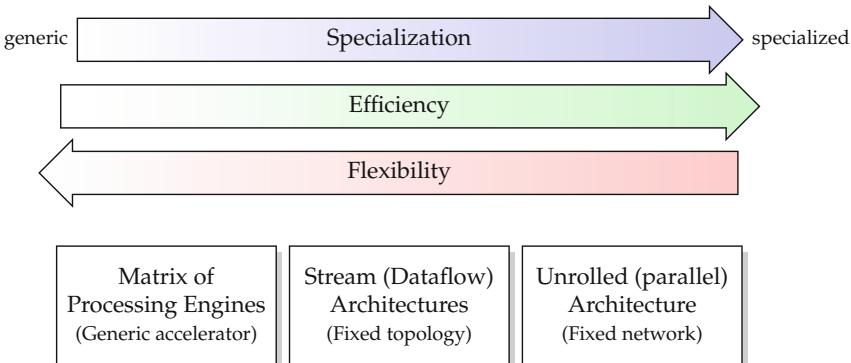


Fig. 24.12 Classification of DNN architectures. (Adopted from Michaela Blott's keynote at Field-Programmable Logic and Applications 2021).

ters (if batch norm is used) are merged into the weights, and the batch norm layers are removed. Next, training continues with QAT to refine the weights to smaller word sizes. The word size can be incrementally reduced, with several training epochs for each word size.

The STE approach works well in practice, but it is not enough to address extreme quantization such as binary or ternary. Here, special methods or STE extensions have been applied for binary [Cou+16; Bet+20] or ternary nets [LZL16; Ale+17; Zhu+16]. The survey in [Bet+20] provides a good overview about current research.

In the case when QAT is not possible (e.g., when training data is proprietary and not available), data-free quantization (DFQ) is an efficient PTQ-based approach that does not depend on the training data but still uses DNN-specific properties during quantization [Jun+21; Guo+22; LBG22].

However, when possible, quantization-aware training provides better results.

24.5 Implementation Aspects

24.5.1 Architectures for Inference

The summary so far is that the arithmetics required to implement deep neural networks mainly consist of multiplication and (multiple-input) addition. Most of the techniques presented in Part I can therefore be used here. Some activation functions may require function approximation techniques reviewed in Part III, but the popular ReLU family again reduces to additions and multiplications. Still, there is an arithmetic challenge in the choice of the number formats to be used for each operation as discussed in Sect. 24.3.

The other main challenge in AI-oriented inference architectures is the organization of these computations. The different DNN architectures are classified in Fig. 24.12 and described next.

In the human brain, each neuron is a hardware one in a 3-D structure. Such a fully parallel architecture can be mimicked in an artificial neural accelerator (right of Fig. 24.12). It is also called an *unrolled* architecture as it can be obtained by unrolling all the loops in the DNN algorithm. Here, each activation in the DNN corresponds to a dedicated physical signal, sometimes registered for high-frequency pipeline operation. It may lead to a very efficient architecture,⁵ but with several drawbacks. The main one is the lack of flexibility, in particular with respect to hyperparameter tuning: changing the number of layers or the size of these layers means changing the hardware.

At the extreme, once the network is trained, all the weights are constant, and most layers become instances of the multiple constant multiplication (MCM) problem of Chap. 12, for which optimized architectures can be built. However, it becomes impossible to change the weights, and therefore such approaches are only considered for reconfigurable hardware. Another major drawback of the fully parallel architecture is its scalability: modern deep neural networks are typically too large to be sensibly implemented in this fully parallel way.

At the other extreme (left of Fig. 24.12), the most generic accelerators perform the inference by using a *matrix of processing engines*. Here the neurons are purely virtual, represented by activations and weights stored in memory. The processing engines perform generic operations, typically vector-matrix, matrix-matrix, or tensor multiplications of a given size. The weights and activations are fed from memory and are sequentially processed. Hence, any DNN can be mapped to such kind of accelerator. The larger the neural network, the longer the time to process one inference.

In such architectures, special care is taken to match the capabilities of the memory hierarchy to the arithmetic computing power, considering the specifics of DNNs. For instance, each weight will be multiplied by many activations, possibly belonging to several independent data in a batch, and an efficient architecture must provide a fast local storage or cache for these weights. This efficient organization of memory access is crucial, but mostly out of the scope of this book. Prominent examples of this type of accelerator (among many others) are NVidia's recent GPU architectures or Google's tensor processing unit (TPU), the latter further presented as one of the case studies in Sect. 24.6.1.

Intermediate architectures between these two extremes exist, which are probably the most promising candidates in the context of application-specific arithmetic. In *stream* or *dataflow* architectures (middle part of Fig. 24.12), a hardware unit is dedicated for each layer, and data is streamed from layer to layer as illustrated in Fig. 24.13. The processing hardware of each layer

⁵ A fundamental limitation is that electronic components can, so far, only be deeply integrated in two dimensions.

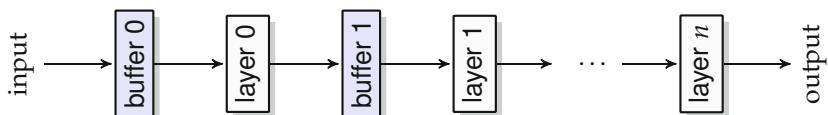


Fig. 24.13 Stream or dataflow architecture for CNN inference.

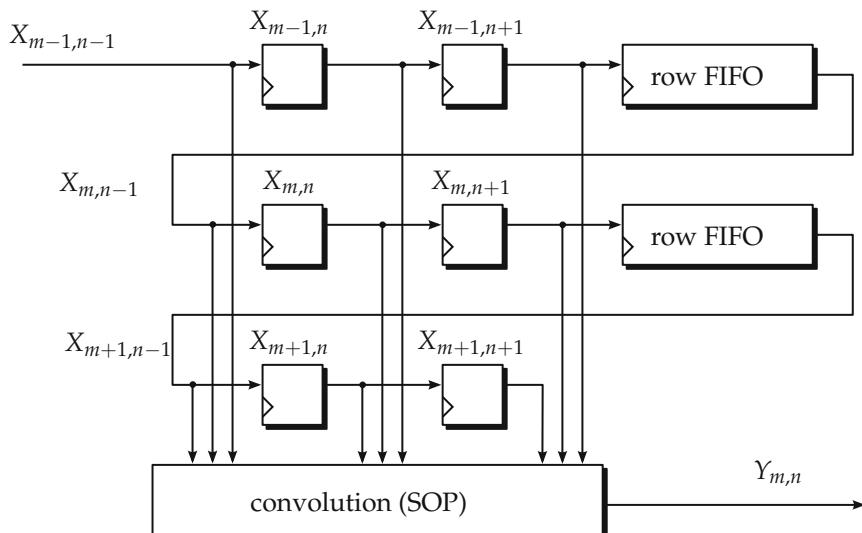


Fig. 24.14 Buffer architecture for a 3×3 convolution in 2D CNN inference.

is specialized to the topology of the given layer. For example, Fig. 24.14 describes the hardware for a layer applying a 3×3 convolution to 2D data. In this example, the 2D data is streamed line by line from top to bottom. Registers and first-in first-out (FIFO) buffers are used so that a 3×3 window of data is available at each clock cycle to be input to a dedicated sum of products (SOP) unit. Control logic (not shown on the figure) is needed to manage the initialization and termination of each buffer and each line, manage zero-padding, etc.

The architecture of Fig. 24.14 computes one activation per cycle. It is possible to replicate it to compute several activations per cycle or to use a sequential SOP implementation that requires less hardware but several clock cycles. The challenge here is to distribute this parallelism between layers to achieve a certain throughput while minimizing the implementation cost of arithmetic and intermediate buffers (remember that different layers may compute different numbers of activations, as illustrated by the volume of the boxes in Fig. 24.8).

24.5.2 Fast Convolution Algorithms

Depending on the architecture classes introduced above, a convolution can be performed

- (a) sequentially, computing **one** result using **several** clock cycles using a multiplier and an accumulator (the latter possibly in larger precision)
- (b) in parallel, computing **one** result in **each** clock cycle using a parallel SOP unit as indicated in Fig. 24.14
- (c) in parallel, computing **several** results in **each** clock cycle. This requires a slightly larger window buffer compared to Fig. 24.14, as the neighboring output values depend on neighboring input values.

In the completely unrolled case, we have one multiplier per weight, so the weights are constant and the methods introduced in Sect. 12.5 can be applied. However, the most common solution is the partly unrolled case (c) above, where the weights are not considered constant. This can be implemented naively by several parallel SOPs. Fast convolution algorithms, reviewed in this section, improve the complexity of this solution.

The size of convolution kernels is typically small, e.g., 3×3 or 5×5 , so the focus here is on convolution algorithms for short convolutions as developed by Toom [Too63] and Cook [Coo66] and later generalized by Winograd [Win80, Chap. VI]. Winograd's fast convolution has been first used for CNNs and extended for multidimensional convolutions by Lavin and Gray [LG16]. We will show the basic ideas here but refer the reader to [LG16], the original work in [Win80] and the excellent book of Blahut [Bla10, Chap. 5].

Winograd's fast convolution algorithms are designed to save multiplications at the cost of extra additions. In fact, these algorithms are minimal with respect to the number of multiplications: it was shown in [Win80] that the minimal number of multiplications required to compute m outputs of a 1-D convolution with r weights requires $n = m + r - 1$ multiplications, which is identical to the number of inputs. Note that a conventional convolution would require $m \times r$ multiplications.

For the general case of a d -dimensional convolution, the number of multiplications is still equal to the number of inputs and becomes

$$n = (m + r - 1)^d, \quad (24.21)$$

instead of $(m \times r)^d$ multiplications in the conventional case. The fast Winograd convolution computing m outputs in parallel of a convolution with r weights, each of dimension d , is usually denoted as $F(m^d, r^d)$. Note that this notation sufficiently describes the number of inputs n as given by (24.21).

Let us start with the smallest example of $F(2, 2)$, i.e., a one-dimensional convolution of $n = 2 + 2 - 1 = 3$ inputs with $r = 2$ weights, where $m = 2$ outputs are computed in parallel. A one-dimensional 2 by 2 convolution can be represented as

$$Y_0 = X_0 W_0 + X_1 W_1 \quad (24.22)$$

$$Y_1 = X_1 W_0 + X_2 W_1, \quad (24.23)$$

which requires four multiplications and two additions. Note that we use a shift in the index compared to (24.7) to simplify the representation. We can rewrite this to

$$Y_0 = M_1 + M_2 \quad (24.24)$$

$$Y_1 = M_2 - M_3 \quad (24.25)$$

with

$$M_1 = (X_0 - X_1) W_0 \quad (24.26)$$

$$M_2 = X_1 (W_0 + W_1) \quad (24.27)$$

$$M_3 = (X_1 - X_2) W_1. \quad (24.28)$$

This only requires three multiplications at the expense of five additions. Thus, one multiplication is replaced by three additions.

Using matrix notation, it can be represented as

$$\mathbf{Y} = \mathbf{C} [(\mathbf{B}\mathbf{X}) \odot (\mathbf{A}\mathbf{W})] \quad (24.29)$$

where \odot denotes the element-wise product, with

$$\mathbf{X} = \begin{pmatrix} X_0 \\ X_1 \\ X_2 \end{pmatrix}, \quad (24.30)$$

$$\mathbf{W} = \begin{pmatrix} W_0 \\ W_1 \end{pmatrix}, \quad (24.31)$$

$$\mathbf{A} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix}, \quad (24.32)$$

$$\mathbf{B} = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & -1 \end{pmatrix}, \quad (24.33)$$

and

$$\mathbf{C} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & -1 \end{pmatrix}. \quad (24.34)$$

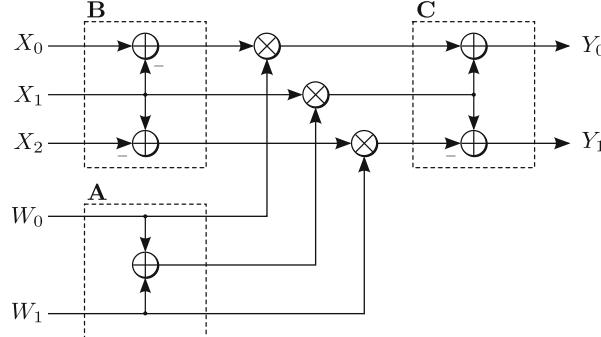


Fig. 24.15 Illustration of the Winograd $F(2,2)$ fast convolution algorithm.

Matrices **A** and **B** describe the necessary pre-additions/subtractions, while **C** describes the post-additions/subtractions as illustrated in Fig. 24.15. Note the similarities to the Karatsuba method used in multipliers (Sect. 8.6, p. 244). In fact, it is a generalization as the Karatsuba method can be described in the same form using matrices **A**, **B** and **C** describing pre-/post-additions.

The minimal 1-D algorithm can be nested within itself to obtain minimal 2-D algorithms (i.e., $F(m^2, r^2)$) using

$$\mathbf{Y} = \mathbf{C} [(\mathbf{B} \mathbf{X} \mathbf{B}^\top) \odot (\mathbf{A} \mathbf{W} \mathbf{A}^\top)] \mathbf{C}^\top, \quad (24.35)$$

where **X** and **W** are now $n \times n$ and $r \times r$ matrices, respectively [LG16]. This nesting can be continued to higher dimensions. We do not detail this here for brevity.

For different n and r , only the matrices are different. We again omit their construction here and refer to [Bla10, Chap. 5]. Let us however highlight the 3×3 kernel which is frequent in CNNs. For two outputs, (i.e., $F(2^d, 3^d)$), the matrices are

$$\mathbf{C} = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{pmatrix}, \quad (24.36)$$

$$\mathbf{B} = \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix}, \quad (24.37)$$

and

Table 24.2 Arithmetic complexity using conventional and Winograd’s fast convolution.

$d m r$	n	Conventional		Winograd	
		Mult	Add	Mult	Add ^a
1 2 2	3	4	2	3	5
1 2 3	4	6	4	4	11 (12)
2 2 2	9	16	12	9	16 (23)
2 2 3	16	36	32	16	77 (92)
$d m r$	$(m + r - 1)^d$	$m^d r^d$	$m^d(r^d - 1)$	$(m + r - 1)^d$	–

^a Numbers in brackets are without considering common subexpression elimination

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & 1 \end{pmatrix}. \quad (24.38)$$

Here all the matrix coefficients in \mathbf{A} , \mathbf{B} and \mathbf{C} are powers of two which can be computed using shifts. However, for larger n and r , some of the coefficients are not powers of two. In the general case, the matrices \mathbf{A} , \mathbf{B} and \mathbf{C} contain rational numbers with small numerators and denominators. Multiplication by these constants can be realized using shift and add (see Sect. 12.4), but this further increases the adder cost.

Table 24.2 compares the arithmetic complexity using conventional and Winograd’s fast convolution for different parameters. It can be observed that the number of multiplications can be reduced significantly at the cost of more adders. Note that some additions can be eliminated, thanks to common subexpression elimination (CSE). For example, the computation of \mathbf{AW} in $F(2, 3)$ results in

$$\left(\frac{\frac{W_0}{2} + \frac{W_1}{2} + \frac{W_2}{2}}{\frac{W_0}{2} - \frac{W_1}{2} + \frac{W_2}{2}} \right). \quad (24.39)$$

The common term $\frac{W_0}{2} + \frac{W_2}{2}$ can be shared, reducing the number of additions from 12 to 11. The last line of Table 24.2 shows the general relations where known [Bla10]. Note that it does not count the cost of the constant multiplications.

Finally, in the context of partly parallelized CNNs, the terms \mathbf{AW} in (24.29) and \mathbf{AWA}^\top in (24.35) only depend on the weights: they can be computed only once and reused many times, e.g., as the convolution is applied to a whole image.

Now for a few arithmetic remarks. The first is that all the previous assumes exact additions (and in general exact multiplication by the matrices **A**, **B** and **C**). However, in this case, the intermediate terms have a larger bit-width, e.g., $X_0 - X_1$ in (24.26) is 1-bit larger than X_0 and X_1 . Therefore, the multipliers in the \odot operation (e.g., the multiplier computing M_1 in (24.26)) are reduced in count, but their cost is slightly higher. An alternative is to round these intermediate terms (and this is what happens when using floating point), but then the fast algorithms may be less accurate than the straightforward ones, and this has to be taken into account. Remember from Sect. 24.3.1 that it is common to keep the full product without rounding it in inference architectures.

Another remark is that for a p -bit fixed-point format, one multiplier costs roughly p adders. The smaller p , the less interesting it is to reduce the multiplier count at the expense of the adder count, all the more as the multipliers become larger.

24.6 Case Studies

To complete this chapter, let us present a small selection of architectures that have been developed for deep neural networks and apply techniques discussed above. The choice of the best architecture for a given application heavily depends on the requirements and constraints of this application. The following selection is necessarily partial (being based on arithmetic considerations more than application-level ones) and soon outdated. These case studies are introduced from the more generic to the more specialized ones (according to Fig. 24.12).

24.6.1 Google's TPU Family

Google's TPU [Jou+17; Jou+18; Jou+21] is a prominent generic AI accelerator (on the left of Fig. 24.12). It is only available in their cloud servers (you can not buy it), but its little sister, the Edge TPU, is sold for embedded AI applications (for a comparison of architectures competing on embedded AI acceleration, see [KJG20]).

The layout for the TPUv1 was presented in detail in [Jou+17], and its evolution over four generations is analyzed in [Jou+21]. TPUv1 was focused on inference, and then v2 and v3 added support for learning. Arithmetically this meant replacing 8-bit integer computations with bfloat16 ones (TPUv4 supports both int8 and bfloat16). Learning also requires more memory and more programmability and is more difficult to parallelize due to possibly diverging weight updates [Jou+21].

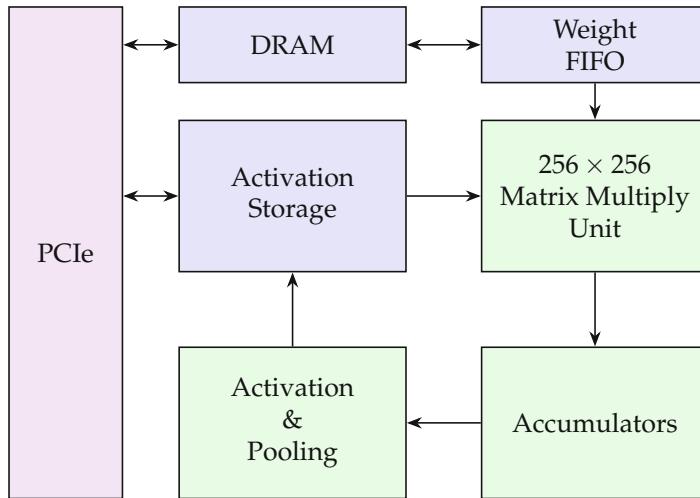


Fig. 24.16 High-level structure of Google’s TPUv1.

The high-level structure of the TPUv1 is shown in Fig. 24.16 where arithmetic parts are shown in green and memories are shown in blue. The TPUv1 was designed as a coprocessor attached to the PCIe I/O bus, similar to a GPU, but it receives instructions over PCIe. The multiply-accumulate (MAC) units are organized as a 256×256 array (65,536 MAC units in total) to form the *Matrix Multiply Unit* [Jou+17] which operates as a systolic array, i.e., in a globally synchronous way.

Each MAC inputs 8-bit signed or unsigned integers. The 16-bit multiplication results can be accumulated in the *Accumulators* unit which can perform 256-element 32-bit accumulations. The TPU can read and write 256 values per clock cycle and can perform either a matrix multiply or a convolution. Hence, it takes B clock cycles to multiply a $B \times 256$ input matrix with a 256×256 constant weight matrix, producing a $B \times 256$ output. An *Activation & Pooling* unit can perform activation functions like ReLU and Sigmoid as well as pooling.

It is possible to use 16-bit precision for either weights or activations, but it will half the speed. Using 16-bit precision for both will quarter the speed.

Note how the weights and activations use different storage units, which also manage their specific access patterns. Weights are stored in an on-chip *Weight FIFO*, fed from an off-chip DDR3 DRAM weight memory, but accessed read-only by the Matrix Multiply Unit. Conversely, the on-chip *Activation Storage* can be read and written to. It is managed by a dedicated direct memory access (DMA) controller.

The instruction set is rather small, containing about a dozen of instructions, like read/write, MatrixMultiply, Convolve or Activate.

Training requires writing to the weight memory as well; therefore, TPUv2 and following revert to a unified on-chip memory for weights and activations [Jou+21].

24.6.2 Binary CNN Architecture

This section summarizes the ideas used within the fully binarized networks [Umu+17] of FINN,⁶ a popular framework for inference on FPGAs. It can be classified as a stream architecture as shown in the middle of Fig. 24.12.

It implements fully binarized networks (see Sect. 24.3.1) with weights and activations in $\{-1, 1\}$. Here, the value -1 is encoded by 0, and the value $+1$ is encoded by 1. The arithmetic is highly customized for that, as first proposed in [KS16; KS18] and used within FINN [Umu+17]. The product of activation and weight can be computed by an XNOR as illustrated by Table 24.3.

Table 24.3 Multiplication in fully binarized nets becomes an XNOR.

Activation code (value)		Weight code (value)		Product code (value)	
0	(-1)	0	(-1)	1	(+1)
0	(-1)	1	(+1)	0	(-1)
1	(+1)	0	(-1)	0	(-1)
1	(+1)	1	(+1)	1	(+1)

The accumulation of the products can be similarly simplified. Consider, for example, the summation of four products $S = P_0 + P_1 + P_2 + P_3$. Table 24.4 illustrates all cases, from all inputs being 0 (representing -1) to all inputs being 1 (representing $+1$) and their sum S . It can be observed that every time a bit flips from 0 to 1, the sum increases by 2. Hence, the value is proportional to the *population count* (or *popcount*) of the inputs. The population count corresponds to the number of bits that are set, i.e.,

$$\text{popcnt} = \sum_i P_i \text{ with } P_i \in \{0, 1\}. \quad (24.40)$$

Technically, `popcnt` can be computed by a *counter* as introduced in Sect. 7.3.2.1, p. 180. Then the sum is $S = 2 \text{popcnt} - N$.

As activation function, the sign function can be seen as a quantized variant of the hyperbolic tangent (compare with Fig. 24.3b, p. 712). To integrate

⁶ <https://xilinx.github.io/finn/>

Table 24.4 Accumulation in fully binarized nets becomes a population count.

P_0	P_1	P_2	P_3	sum S	popcnt
0 (-1) 0 (-1) 0 (-1) 0 (-1)				-4	0
0 (-1) 0 (-1) 0 (-1) 1 (+1)				-2	1
0 (-1) 0 (-1) 1 (+1) 1 (+1)				0	2
0 (-1) 1 (+1) 1 (+1) 1 (+1)				2	3
1 (+1) 1 (+1) 1 (+1) 1 (+1)				4	4

the batch normalization parameters as discussed in Sect. 24.2.3, it can be generalized to a threshold function. The threshold value is specific to each neuron. The threshold function is computed by an unsigned comparator (see Chap. 6) on the popcnt value [Umu+17] and delivers the activations of the next layer.

Overall, high classification rates with small power consumption and latency were achieved at the cost of a reduced accuracy compared to floating-point CNNs.

24.6.3 AddNet: FPGA-Specific Optimization of the Multipliers

This section presents another stream architecture [Far+20] where the weights are quantized to a set of values that (1) Matches the distribution of the CNN to be implemented (2) Have an efficient implementation as a reconfigurable constant coefficient multiplier (RCCM) (see Sect. 12.6.2)

Different methods exist to design an reconfigurable constant coefficient multiplier (RCCM) that realizes a given set of coefficients [DKD07; THP07; Möl+17]. The idea in [Far+20] was to do the reverse: RCCM topologies were searched that produce the largest number of *useful* coefficients for a given adder complexity. The *useful* coefficients are those which approximately match the weight distribution appearing in the neural network. More precisely, the possible coefficient sets for a given RCCM topology using a certain number of adders are enumerated, and the set with the closest distribution (measured by the Kullback-Leibler divergence [KL51]) is selected.

The typical weight distribution in the neural net has a Gauss-like distribution, so numbers around zero should appear more frequently than larger numbers. Figure 24.17a shows the average distribution of weights in the popular CNNs AlexNet [KSH12] and MobileNet [How+17].

An example RCCM obtained by this method is shown in Fig. 24.18. This work targets FPGAs, and as an additional target-specific optimization, the only multiplexer sizes considered were those that fit into the same basic

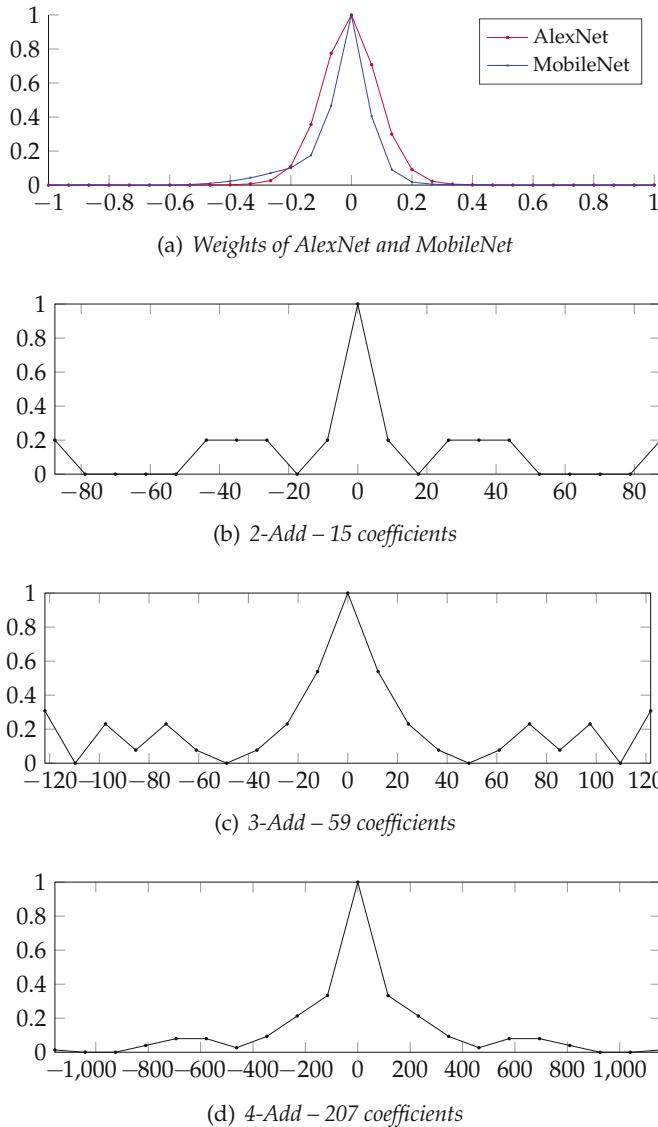


Fig. 24.17 Distribution for CNN weights and constant multiplier coefficients.

logic element (BLE) that is used to construct the ripple-carry adder (see Sect. 5.4.3). By doing so, the additional MUX(es) come without extra cost. For instance, the whole circuit of Fig. 24.18 consumes the same resources as two adders (and is therefore called 2-Add RCCM in the following). As Table 24.5 illustrates, it can multiply by 15 different coefficients, which are configured using the select lines s_0 to s_3 .

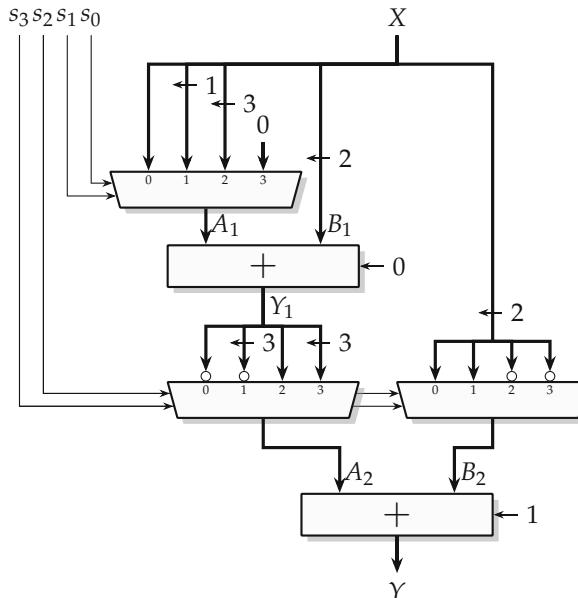


Fig. 24.18 2-Add RCCM topology for the coefficients $\pm\{0, 1, 2, 8, 28, 36, 44, 92\}$.

This idea was evaluated for RCCMs that utilize 2, 3, and 4 adders (called 2-Add, 3-Add, and 4-Add RCCMs). The closest coefficient sets obtained for the AlexNet model are given in Table 24.6. Their distributions are shown in Fig. 24.17b, c and d. They are similar to that of the pretrained model shown in Fig. 24.17a.

Using a variant of quantization-aware training (see Sect. 24.4.4), the weights can be trained to the closest fixed-point representations that correspond to the RCCM integers listed in Table 24.6. Using an activation word size of 8 bits, a top-1 accuracy close to the floating-point baseline could be achieved, outperforming nets using binary or ternary weights with a relatively small increase of LUT resources (about 20%) [Far+20].

24.6.4 Unrolling a Ternary CNN

This case study attempts to implement an image classification CNN that can process one pixel of the input image each cycle, leading to one classification every $A_x^0 \times A_y^0$ clock cycles (the input image size). A fully unrolled network would process one image per cycle, but this is currently unfeasible considering the size of modern networks. It can hence be classified as a stream architecture (middle part of Fig. 24.12) but unrolled in such a way to exploit

Table 24.5 Truth table of the RCCM of Fig. 24.18.

s_3	s_2	s_1	s_0	A_1/X	B_1/X	Y_1/X	A_2/X	B_2/X	Y/X
0	0	0	0	1	4	5	-5	4	-1
0	0	0	1	2	4	6	-6	4	-2
0	0	1	0	8	4	12	-12	4	-8
0	0	1	1	0	4	4	-4	4	0
0	1	0	0	1	4	5	-40	4	-36
0	1	0	1	2	4	6	-48	4	-44
0	1	1	0	8	4	12	-96	4	-92
0	1	1	1	0	4	4	-32	4	-28
1	0	0	0	1	4	5	5	-4	1
1	0	0	1	2	4	6	6	-4	2
1	0	1	0	8	4	12	12	-4	8
1	0	1	1	0	4	4	4	-4	0
1	1	0	0	1	4	5	40	-4	36
1	1	0	1	2	4	6	48	-4	44
1	1	1	0	8	4	12	96	-4	92
1	1	1	1	0	4	4	32	-4	28

Table 24.6 Optimized RCCM coefficients.

arch.	#coeff	Coefficient set (\pm)
2-Add	15	0 1 2 8 28 36 44 92
3-Add	59	0 1 2 3 4 5 6 7 9 10 12 13 14 16 23 29 30 32 63 69 70 72 87 93 94 96 119 125 126 128
4-Add	207	0 1 2 4 5 7 8 9 11 13 14 15 16 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 36 37 38 39 40 46 48 54 58 64 69 70 71 74 75 76 78 80 81 82 84 85 87 94 96 102 114 118 126 134 142 150 166 174 182 190 194 198 206 214 222 230 238 246 258 262 270 278 286 302 310 318 326 334 382 398 446 450 526 566 574 582 614 622 654 662 670 686 694 710 766 782 830 1214

one opportunity: to have dedicated hardware for each set of weights that appears in the unrolled network, i.e., each convolution kernel in the early layers, and each fully connected neuron in the later layers (see Fig. 24.8). This enables two types of optimizations:

- Zero weights directly entail that no hardware is needed for the corresponding multiplications.
- Nonzero weights can be optimized using constant multiplication techniques, for instance, those reviewed in Chap. 12.

Still, this level of parallelism is very challenging to realize for real-size CNNs. For illustration, our case study is based on a modified version of the ternary CNN of [LZL16] called VGG-7, itself based on the VGG family introduced in [SZ15] for image classification. Its structure was already shown

in full detail in Fig. 24.8. This network was trained on the CIFAR-10 dataset⁷ which contains 60,000 32×32 color images of 10 different classes like airplanes, cars, birds, cats, etc. Table 24.7 shows the input and output size of each layer and the corresponding complexity in MAC operations. For example in layer 1, there are $3 \times 3 \times 3 = 27$ inputs (the kernel size) that have to be convolved by 64 filter kernels to obtain the 64 activations of layer 2. This corresponds to a matrix size of $27 \times 64 = 1728$ operations. One can observe that the operation count is heavily growing for the later layers, leading to over one million MAC operations in total.

A first step toward addressing this complexity is to use ternary weights [Tri+19]. Thus, the MAC operations correspond to additions or subtractions (as activations are only multiplied by 1 or -1 in a ternary CNN). However, even without the costly multiplications, a direct mapping to an FPGA remains impossible. For example, the rather large Ultrascale+ FPGA VU9P used in [Tri+19] provides about 1.2 million LUTs. Using 16 bits for the activations would require 16 LUTs to realize one adder, the 1.14 million operations of Table 24.7 would require about 18 million LUTs.

A second step is to train the network to maximize the number of zero weights, since each zero weight will save one 16-bit addition. For this sparse training, the framework of [LZL16] uses QAT with a threshold for the quantization. Any floating-point weight below this threshold is quantized to 0; all other values are quantized to ± 1 , depending on their sign. This threshold directly controls the sparsity of the solution and was carefully tuned for each layer to get a maximally sparse solution with high accuracy.

Table 24.7 Arithmetic complexity of the VGG-7 CNN layers.

Layer	Inputs		Outputs	Operations
1	$3 \times 3 \times 3 = 27$		64	$27 \times 64 = 1,728$
2	$3 \times 3 \times 64 = 576$		64	$576 \times 64 = 36,864$
3	$3 \times 3 \times 64 = 576$		128	$576 \times 128 = 73,728$
4	$3 \times 3 \times 128 = 1152$		128	$1152 \times 128 = 147,456$
5	$3 \times 3 \times 128 = 1152$		256	$1152 \times 256 = 294,912$
6	$3 \times 3 \times 256 = 2304$		256	$1152 \times 256 = 589,824$
				Total: 1,144,512

The last optimization step is to develop a hardware generation algorithm specific to the resulting constant matrices. It is a constant matrix-vector multiplication (see Sect. 12.5.4), but the matrices only have elements of $\{-1, 0, 1\}$, so there are no redundancies between the constant coefficients (they do not have common factors). However, there is redundancy between the output elements, i.e., a common term like $X_1 + X_5$ may appear in several

⁷ <https://www.cs.toronto.edu/~kriz/cifar.html>

rows. Therefore, the graph-based reduced pipelined adder graph (RPAG) algorithm [Kum+12] can be used, but a common subexpression elimination (CSE) algorithm also fits well to this problem and scales better with the number of outputs. Hence, for the first layers, RPAG was used, and extended versions of the CSE heuristics proposed in [HCT06; Wu+13] were used in the other layers. RPAG's extension for ternary (3-input) adders [Har+18] was used in the first layer to fully utilize the logic of the target FPGA.

Since the hardware iterates over the input pixels, it is still a streaming architecture, and we now come to the usual question of balancing the data rates between layers. Indeed, the data rate is decreasing with each pooling layer. The first pooling layer reduces the rate by 1/4; after the second pooling, the rate is only 1/16, etc. In a traditional streaming architecture this is compensated by the increase of the number of channels, but not here since all the channels are computed in parallel (since there is a set of weights per channel).

One idea would be to share the arithmetic, but it is not compatible with the exploitation of sparsity and redundancy as described above. Another solution, more arithmetic, is to use digit-serial adders/subtractors. As Fig. 24.19 shows, these are simple and trade cycles for logic resources. After the first pooling, data can take four cycles to be processed using the radix-16 digit serial adder of Fig. 24.19a. After the second pooling, the 16-cycle bit-sequential adder of Fig. 24.19b can be used.

Note that parallel-in/sequential-out shift registers are required after each pooling. Afterward, every word is processed in a digit-wise manner.

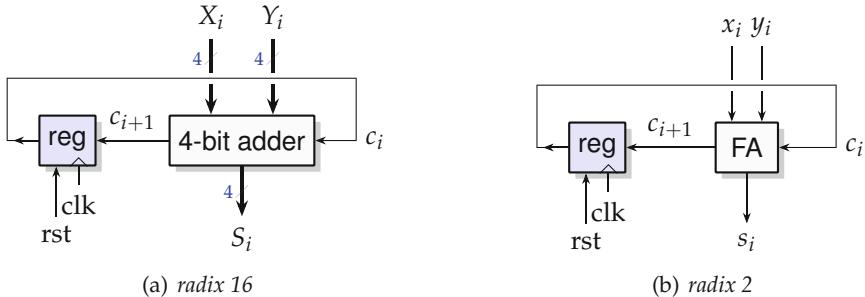


Fig. 24.19 Digit-serial adders take more cycles but consume fewer logic resources.

Overall, all the convolutional layer could be mapped to the Ultrascale+ VU9P FPGA of an Amazon's AWS F1 instance, utilizing 66.6% and 41.6% of the LUTs and flip-flops (FFs), respectively, and reaching 122 k frames per second with a latency of just 30 μ s.

It is interesting to compare this work to the work in [PBP18], which uses ternary weights *and activations* (except for the first layer which inputs 8-bit RGB data). It does not consider the weights constant and from this point of

view is a more sequential data-flow architecture. Its throughput is slightly lower (60 k frames/s on the previous generation technology), but changing the weights does not require a complete recompilation of the architecture. This illustrates the trade-off shown in Fig. 24.12.

24.6.5 LogicNets

Compared to the previous section, LogicNets [Umu+20] goes one step further by packing the complete functionality of a neuron (sum of products by constants (SOPC), accumulation, and activation function) into a single table. As every neuron is mapped to an equivalent table running in parallel, we can classify it as an unrolled architecture targeting high-throughput applications (the right of Fig. 24.12, p. 734).

The storage requirements of the table only depend on the input and output word size. The implementation complexity roughly scales with its storage requirements. As a complete neuron is mapped to the table, including the activation function that generates a quantized output, its internal complexity does not matter. An interesting property here is that all internal weights can be kept as floating-point weights, so no quantization aware-training has to be performed. Batch normalization can also be included without having to care about merging the batch norm parameters back into the weights. Figure 24.20 shows this for a binarized 6-input neuron producing a single bit output in a single LUT6.

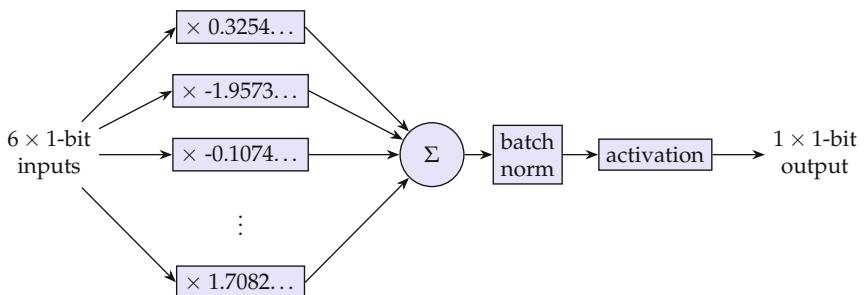


Fig. 24.20 Structure of a LogicNet neuron.

However, typical neurons have more inputs and/or more bits per input/output. As the complexity grows exponentially with the number of input bits, the fan-in of each neuron has to be limited. This is done by restricting the number of connections to previous neurons (by imposing sparsity on the network), as well as limiting the activation word size to <4 bits. The training is performed by a training algorithm for sparse CNNs proposed in [PVN18].

A design space exploration was used to find good trade-offs between accuracy and complexity (estimated using a rather pessimistic estimation). The performed experiments using up to seven activation inputs per neuron using an activation word size of 1 to 3 bits show some interesting results (after synthesis): The accuracy of these low-precision networks mainly depends on the precision of the activations, and it seems hard to impossible to reach the same accuracy by making the network larger. For example, even using about ten times more neurons in a binary net did not reach the accuracy of a 2-bit network. This shows that practical training methods currently do not reach what is expected from theoretic results [Con20]. Overall, the LogicNets idea is good for small machine learning tasks that require extreme throughput.

References

- [ACJ20] Mark Arnold, Ed Chester, and Corey Johnson. "Training Neural Nets using only an Approximate Tableless LNS ALU". In: *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2020, pp. 69–72. (cit. on pp. [727](#), [730](#)).
- [Agg18] Charu C. Aggarwal. *Neural Networks and Deep Learning*. Springer, 2018. (cit. on pp. [708](#), [731](#), [732](#), [733](#)).
- [AGG21] Syed Asad Alam, James Garland, and David Gregg. "Low-Precision Logarithmic Number Systems: Beyond Base-2". In: *ACM Transactions on Architecture and Code Optimization* 18.4 (2021), pp. 1–25. (cit. on p. [727](#)).
- [Agr+19] Ankur Agrawal, Silvia M. Mueller, Bruce M. Fleischer, Jungwook Choi, Naigang Wang, Xiao Sun, and Kailash Gopalakrishnan. "DLFloat: A 16-b Floating Point format designed for Deep Learning Training and Inference". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2019, pp. 92–95. (cit. on p. [729](#)).
- [Ale+17] Hande Alemdar, Vincent Leroy, Adrien Prost-Boucle, and Frédéric Pérot. "Ternary Neural Networks for Resource-Efficient AI Applications". In: *2017 international joint conference on neural networks (IJCNN)*. IEEE. 2017, pp. 2547–2554. (cit. on pp. [726](#), [727](#), [734](#)).
- [Alt05] *FFT/IFFT Block Floating Point Scaling*. Application note 404-1.0. Altera Corporation. 2005. (cit. on p. [726](#)).
- [And+18] Kota Ando, Kodai Ueyoshi, Kentaro Orimo, Haruyoshi Yonekawa, Shimpei Sato, Hiroki Nakahara, Shinya Takamaeda-Yamazaki, Masayuki Ikebe, Tetsuya Asai, Tadahiro Kuroda, and Masato Motomura. "BRein Memory: A Single-Chip Binary/Ternary Reconfigurable in-Memory Deep Neural Network

- Accelerator Achieving 1.4 TOPS at 0.6 W". In: *IEEE Journal of Solid-State Circuits* 53.4 (2018), pp. 983–994. (cit. on p. [726](#)).
- [Bet+20] Joseph Bethge, Christian Bartz, Haojin Yang, Ying Chen, and Christoph Meinel. "MeliusNet: Can Binary Neural Networks Achieve MobileNet-level Accuracy?" 2020. arXiv: 2001 . 05936v2. (cit. on pp. [726](#), [727](#), [734](#)).
- [Bla10] Richard E. Blahut. *Fast Algorithms for Signal Processing*. Cambridge University Press, 2010. (cit. on pp. [737](#), [739](#), [740](#)).
- [BLC13] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. "Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation". 2013. arXiv: 1308.3432v1 [cs.LG]. (cit. on p. [733](#)).
- [BPL10] Y-Lan Boureau, Jean Ponce, and Yann LeCun. "A Theoretical Analysis of Feature Pooling in Visual Recognition". In: *International Conference on Machine Learning*. 2010, pp. 111–118. (cit. on p. [719](#)).
- [Bur+19] Neil Burgess, Nigel Stephens, Jelena Milanovic, and Konstantinos Monachopoulos. "Bfloat16 Processing for Neural Networks". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2019, pp. 88–91. (cit. on p. [729](#)).
- [CBD15] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. "BinaryConnect: Training Deep Neural Networks with binary weights during propagations". In: *Advances in neural information processing systems* 28 (2015). (cit. on p. [726](#)).
- [CDP22] Maxime Christ, Florent de Dinechin, and Frédéric Pétrot. "Low-precision logarithmic arithmetic for neural network accelerators". In: *International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE, 2022. (cit. on pp. [713](#), [727](#), [728](#)).
- [Chm+22] Brian Chmiel, Ron Banner, Elad Hoffer, Hilla Ben Yaacov, and Daniel Soudry. "Logarithmic Unbiased Quantization: Simple 4-bit Training in Deep Learning". 2022. arXiv: 2112 . 10769v2. (cit. on p. [730](#)).
- [Con20] George A Constantinides. "Rethinking arithmetic for deep neural networks". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378.2166 (2020), p. 20190051. (cit. on pp. [727](#), [751](#)).
- [Coo66] Stephen Arthur Cook. "On the minimum computation time of functions". PhD thesis. Harvard University, Cambridge, Mass., 1966. (cit. on p. [737](#)).
- [Cou+16] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran ElYaniv, and Yoshua Bengio. "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1". 2016. arXiv: 1602 . 02830v3 [cs.LG]. (cit. on pp. [727](#), [734](#)).

- [Den+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. "ImageNet: A Large-Scale Hierarchical Image Database". In: *Computer Vision and Pattern Recognition Work-shops*. IEEE, 2009, pp. 248–255. (cit. on p. 730).
- [DKD07] Süleyman S. Demirsoy, Izzet Kale, and Andrew Dempster. "Re-configurable Multiplier Blocks: Structures, Algorithm and Applications". In: *Circuits, Systems, and Signal Processing* 26.6 (2007), pp. 793–827. (cit. on p. 744).
- [Far+20] Julian Faraone, Martin Kumm, Martin Hardieck, Peter Zipf, Xueyuan Liu, David Boland, and Philip H. W. Leong. "AddNet: Deep Neural Networks Using FPGA-Optimized Multipliers". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28.1 (2020), pp. 115–128. pp. 744, 746). (cit. on pp. 744, 746).
- [FH19] Matthias Feurer and Frank Hutter. "Hyperparameter Optimization". In: *Automated Machine Learning*. Springer, 2019, pp. 3–33. (cit. on p. 713).
- [GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks". In: *International Conference on Artificial Intelligence and Statistics*. Vol. 15. 2011, pp. 315–323. (cit. on p. 733).
- [GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. (cit. on pp. 707, 708, 712, 713, 719, 722, 731, 732).
- [Gho+21] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. "A Survey of Quantization Methods for Efficient Neural Network Inference". 2021. arXiv: 2103.13630. (cit. on p. 733).
- [Guo+22] Cong Guo, Yuxian Qiu, Jingwen Leng, Xiaotian Gao, Chen Zhang, Yunxin Liu, Fan Yang, Yuhao Zhu, and Minyi Guo. "SQuant: On-the-Fly Data-Free Quantization via Diagonal Hessian Approximation". In: *International Conference on Learning Representations (ICLR)*. 2022. (cit. on p. 734).
- [Gys+18] Philipp Gysel, Jon Pimentel, Mohammad Motamedi, and Soheil Ghiasi. "Ristretto: A Framework for Empirical Study of Resource-Efficient Inference in Convolutional Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 29.11 (2018), pp. 5784–5789. (cit. on p. 726).
- [Gys16] Philipp Gysel. "Ristretto: Hardware-Oriented Approximation of Convolutional Neural Networks". MA thesis. University of California, 2016. arXiv: 1605.06402. (cit. on p. 726).
- [Hah+00] Richard H. R. Hahnloser, Rahul Sarpeshkar, Misha A. Mahowald, Rodney J. Douglas, and H. Sebastian Seung. "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit". In: *Nature* 405.6789 (2000), pp. 947–951. (cit. on p. 712).

- [Har+18] Martin Hardieck, Martin Kumm, Patrick Sittel, and Peter Zipf. "Constant Matrix Multiplication with Ternary Adders". In: *International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, 2018, pp. 85–88. (cit. on p. [749](#)).
- [HCT06] Shen-Fu Hsiao, Ming-Chih Chen, and Chia-Shin Tu. "Memory-Free Low-Cost Designs of Advanced Encryption Standard Using Common Subexpression Elimination for Subfunctions in Transformations". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 53.3 (2006), pp. 615–626. (cit. on p. [749](#)).
- [He+16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition". In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2016, pp. 770–778. (cit. on pp. [708](#), [721](#)).
- [HG16] Dan Hendrycks and Kevin Gimpel. "Gaussian Error Linear Units (GELUs)". 2016. arXiv: 1606.08415v4 [cs.LG]. (cit. on p. [713](#)).
- [How+17] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". 2017. arXiv: 1704.04861. (cit. on p. [744](#)).
- [HS97] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780. (cit. on p. [709](#)).
- [HTH19] Greg Henry, Ping Tak Peter Tang, and Alexander Heinecke. "Leveraging the bfloat16 Artificial Intelligence Datatype For Higher-Precision Computations". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2019, pp. 97–98. (cit. on p. [729](#)).
- [IS15] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *32nd International Conference on Machine Learning*. 2015, pp. 448–456. (cit. on p. [721](#)).
- [Jac+18] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018. (cit. on p. [726](#)).
- [Joh18] Jeff Johnson. "Rethinking floating point for deep learning". 2018. arXiv: 1811.01721v1. (cit. on p. [727](#)).
- [Jou+17] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *International Symposium on Computer Architecture (ISCA)*. 2017, pp. 1–12. (cit. on pp. [741](#), [742](#)).

- [Jou+18] Norman P. Jouppi, Cliff Young, Nishant Patil, and David Patterson. "A Domain-Specific Architecture for Deep Neural Networks". In: *Communications of the ACM* (2018). (cit. on p. 741).
- [Jou+21] Norman P. Jouppi et al. "Ten Lessons From Three Generations Shaped Google's TPUv4i". In: *International Symposium on Computer Architecture (ISCA)*. 2021, pp. 1–14. (cit. on pp. 741, 743).
- [Jun+21] Youngbeom Jung, Hyeonuk Kim, Yeongjae Choi, and Lee-Sup Kim. "Quantization-Error-Robust Deep Neural Network for Embedded Accelerators". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* (2021), pp. 1–1. (cit. on pp. 726, 734).
- [Kim+18] Min Soo Kim, Alberto A. Del Barrio, Román Hermida, and Nader Bagherzadeh. "Low-power Implementation of Mitchell's Approximate Logarithmic Multiplication for Convolutional Neural Networks". In: *Design Automation Conference (DAC)*. ACM/IEEE, 2018, pp. 617–622. (cit. on p. 727).
- [KJG20] Luke Kljucaric, Alex Johnson, and Alan D. George. "Architectural Analysis of Deep Learning on Edge Accelerators". In: *High Performance Extreme Computing Conference (HPEC)*. 2020, pp. 1–7. (cit. on p. 741).
- [KL51] Solomon Kullback and Richard A Leibler. "On Information and Sufficiency". In: *The Annals of Mathematical Statistics* 22.1 (1951), pp. 79–86. (cit. on p. 744).
- [KP20] Ioannis Kouretas and Vassilis Palouras. "Hardware Implementation of a Softmax-Like Function for Deep Learning". In: *Technologies* 8.3 (2020), p. 46. (cit. on p. 722).
- [KS16] Minje Kim and Paris Smaragdis. "Bitwise Neural Networks". 2016. arXiv: 1601.06071. (cit. on p. 743).
- [KS18] Minje Kim and Paris Smaragdis. "Bitwise Neural Networks for Efficient Single-Channel Source Separation". In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2018, pp. 701–705. (cit. on pp. 726, 743).
- [KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *International Conference on Neural Information Processing Systems*. Curran Associates Inc., 2012, pp. 1097–1105. (cit. on pp. 708, 744).
- [Kum+12] Martin Kumm, Peter Zipf, Mathias Faust, and Chip-Hong Chang. "Pipelined Adder Graph Optimization for High Speed Multiple Constant Multiplication". In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2012, pp. 49–52. (cit. on p. 749).
- [LBG22] Cecilia Latotzke, Batuhan Balim, and Tobias Gemmeke. "Post-Training Quantization for Energy Efficient Realization of Deep Neural Networks". In: *arXiv* (2022). eprint: 2210.07906. (cit. on p. 734).

- [LeC19] Yann LeCun. "Deep Learning Hardware: Past, Present, and Future". In: *International Solid-State Circuits Conference(ISSCC)*. IEEE, 2019, pp. 12–19. (cit. on pp. [713](#), [724](#)).
- [Lee+17] Edward H Lee, Daisuke Miyashita, Elaina Chai, Boris Murmann, and S. Simon Wong. "Lognet: Energy-efficient neural networks using logarithmic computation". In: *International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2017, pp. 5900–5904. (cit. on p. [727](#)).
- [LG16] Andrew Lavin and Scott Gray. "Fast Algorithms for Convolutional Neural Networks". In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2016, pp. 4013–4021. (cit. on pp. [737](#), [739](#)).
- [LZL16] Fengfu Li, Bo Zhang, and Bin Liu. "Ternary Weight Networks". 2016. arXiv: 1605.04711 [cs.CV]. 727, 734, 747, 748). (cit. on pp. [722](#), [726](#), [727](#), [734](#), [747](#), [748](#)).
- [Mic+22] Paulius Micikevicius, Dusan Stosic, Patrick Judd, John Kamalu, Stuart Oberman, Mohammad Shoeybi, Michael Siu, Hao Wu, Neil Burgess, Sangwon Ha, Richard Grisenthwaite, Naveen Mellemudi, Marius Cornea, Alexander Heinecke, and Pradeep Dubey. "FP8 Formats for Deep Learning". 2022. arXiv: 2209.05433. (cit. on p. [729](#)).
- [MLM16] Daisuke Miyashita, Edward H. Lee, and Boris Murmann. "Convolutional Neural Networks Using Logarithmic Data Representation". arXiv preprint arXiv:1603.01025. 2016. (cit. on p. [727](#)).
- [Möl+17] Konrad Möller, Martin Kumm, Marco Kleinlein, and Peter Zipf. "Reconfigurable Constant Multiplication for FPGAs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36.6 (2017), pp. 927–937. (cit. on p. [744](#)).
- [NH10] Vinod Nair and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *International Conference on Machine Learning*. 2010. (cit. on p. [712](#)).
- [Nou+22] Badreddine Noune, Philip Jones, Daniel Justus, Dominic Masters, and Carlo Luschi. "8-bit Numerical Formats for Deep Neural Networks". 2022. arXiv: 2206.02915. (cit. on p. [729](#)).
- [NVI20] NVIDIA. *NVIDIA A100 Tensor Core GPU Architecture*. Tech. rep. 2020. (cit. on p. [729](#)).
- [PBP18] Adrien Prost-Boucle, Alban Bourge, and Frédéric Pétrot. "High-efficiency Convolutional Ternary Neural Networks With Custom Adder Trees and Weight Compression". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11.3 (2018), pp. 1–24. (cit. on pp. [726](#), [749](#)).
- [PVN18] Ameya Prabhu, Girish Varma, and Anoop Namboodiri. "Deep Expander Networks: Efficient Deep Networks from Graph Theory". In: *European Conference on Computer Vision (ECCV)*. 2018. (cit. on p. [750](#)).

- [Qin+20] Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. "Binary neural networks: A survey". In: *Pattern Recognition* 105 (2020), p. 107281. (cit. on p. 727).
- [Ras+16] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks". 2016. arXiv: 1603 . 05279. (cit. on p. 726).
- [Red+16] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. "You Only Look Once: Unified, Real-Time Object Detection". In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2016, pp. 779–788. (cit. on pp. 708, 721).
- [RF16] Joseph Redmon and Ali Farhadi. "YOLO9000: Better, Faster, Stronger". 2016. arXiv: 1612.08242. (cit. on p. 721).
- [RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (1986), pp. 533–536. (cit. on pp. 714, 731).
- [Shi+17] Kyuhong Shim, Minjae Lee, Iksoo Choi, Yoonho Boo, and Wonyong Sung. "SVD-Softmax – Fast Softmax Approximation on Large Vocabulary Neural Networks." In: *International Conference on Neural Information Processing Systems (NIPS)*. 2017. (cit. on p. 722).
- [Sun+19] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Viji Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. "Hybrid 8-bit Floating Point (HFP8) Training and Inference for Deep Neural Networks". In: *Advances in neural information processing systems* 32 (2019). (cit. on p. 729).
- [SZ15] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2015. (cit. on pp. 722, 747).
- [Sze+17] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey". In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329. (cit. on pp. 707, 708).
- [Tam+19] Thierry Tambe, En-Yu Yang, Zishen Wan, Yuntian Deng, Vijay Janapa Reddi, Alexander Rush, David Brooks, and Gu-Yeon Wei. "AdaptivFloat: A Floating-point based Data Type for Resilient Deep Learning Inference". 2019. arXiv: 1909.13271. (cit. on p. 729).
- [Tan+17] Hokchhay Tann, Soheil Hashemi, R. Iris Bahar, and Sherief Reda. "Hardware-Software Codesign of Accurate, Multiplier-Free Deep Neural Networks". In: *Design Automation Conference (DAC)*. ACM/IEEE, 2017. (cit. on p. 727).

- [THP07] Peter Tummeltshammer, James C. Hoe, and Markus Püschel. “Time-Multiplexed Multiple-Constant Multiplication”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.9 (2007), pp. 1551–1563. (cit. on p. [744](#)).
- [TL19] Mingxing Tan and Quoc Le. “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks”. In: *36th International Conference on Machine Learning*. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 6105–6114. (cit. on p. [713](#)).
- [Too63] Andrei Leonovich Toom. “The complexity of a scheme of functional elements realizing the multiplication of integers”. In: *Soviet Mathematics-Doklady* 7 (1963), pp. 714–716. (cit. on p. [737](#)).
- [Tri+19] Stephen Tridgell, Martin Kumm, Martin Hardieck, David Boland, Duncan Moss, Peter Zipf, and Philip H. W. Leong. “Unrolling Ternary Neural Networks”. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 12.4 (2019), pp. 1–23. (cit. on pp. [722](#), [726](#), [727](#), [748](#)).
- [Umu+17] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Visser. “FINN: A Framework for Fast, Scalable Binarized Neural Network Inference”. In: *International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2017, pp. 65–74. (cit. on pp. [726](#), [727](#), [743](#), [744](#)).
- [Umu+20] Yaman Umuroglu, Yash Akhauri, Nicholas James Fraser, and Michaela Blott. “LogicNets: Co-Designed Neural Networks and Circuits for Extreme-Throughput Applications”. In: *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2020, pp. 291–297. (cit. on p. [750](#)).
- [VKB18] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. “Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions”. In: *ACM Computing Surveys (CSUR)* 51.3 (2018), pp. 56–39. (cit. on p. [708](#)).
- [Win80] Shmuel Winograd. *Arithmetic complexity of computations*. Vol. 33. BMS-NSF Regional Conference Series in Applied Mathematics. SIAM, 1980. (cit. on p. [737](#)).
- [Wol+20] Thomas Wolf et al. “Transformers: State-of-the-art natural language processing”. In: *Conference on empirical methods in natural language processing: system demonstrations*. Association for Computational Linguistics, 2020, pp. 38–45. (cit. on p. [709](#)).
- [Wu+13] Ning Wu, Xiaoqiang Zhang, Yunfei Ye, and Lidong Lan. “Improving Common Subexpression Elimination Algorithm with a New Gate-Level Delay Computing Method”. In: *World Congress on Engineering and Computer Science*. 2013. (cit. on p. [749](#)).

- [Zha+21] Jiawei Zhao, Steve Dai, Rangharajan Venkatesan, Ming-Yu Liu, Brucek Khailany, Bill Dally, and Anima Anandkumar. “LNS-Madam: Low-Precision Training in Logarithmic Number System using Multiplicative Weight Update”. 2021. arXiv: 2106.13914. (cit. on p. [730](#)).
- [Zhu+16] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. “Trained Ternary Quantization”. In: *International Conference on Learning Representations*. 2016, pp. 1–10. (cit. on pp. [726](#), [727](#), [734](#)).

APPENDIX A

Custom Arithmetic Datapath Design with FloPoCo

When FPGAs are better at floating point than microprocessors.

Not your neighbor's FPU.

All the operators you will never see in a microprocessor.

FPGA arithmetic the way it should be.

Circuits computing just right.

Fantastic arithmetic beasts (and how to build them).

Some of the successive advertising phrases for the FloPoCo project

The FloPoCo software project embodies much of the spirit of Application-Specific Arithmetic as discussed in this book. This appendix attempts to complement the software documentation with a guided tour of this software’s history and philosophy, both for users and prospective developers.

A.1 History of the FloPoCo Project

The FloPoCo open-source arithmetic core generator project started modestly in 2008 [DKP09] with a few parametric **floating point cores**. Since then, it has evolved to become a framework for research on hardware arithmetic cores at large, including among others: LNS arithmetic [VCA10], random number generators [Tho13], elementary functions [DJP10; DP10; Din+13; DIS13; DI15; Tho15], specialized operators such as constant multiplication and division [KLZ12; Din12; Kum+16; Uğ+17], various FPGA-specific optimization techniques [NPP11; Bru+13; ID17], posit arithmetic [MDB20], and more recently signal-processing transforms and filters [Vol+19; GMK19] and neural networks [CDP22; Hab+22]. It is also used as backend (for

the VHDL code generation) in the Origami high-level synthesis (HLS) tool [Kum+22; Sit+17a; Sit+17b; Sit+19; FSZ22] and the neural network code generator Elysia [Har+22]. Some of this research has been synthesized in this book, and more references can be found on the web site of the project: <http://flopoco.org/>.

A.2 FloPoCo From a User Point of View

We refer the user to <http://flopoco.org/> for instructions on downloading, installing, and running FloPoCo. FloPoCo is written in C++ and depends on several powerful libraries. For just generating operators or trying out FloPoCo, an experimental web interface is available from this web site.

A.2.1 Overview

FloPoCo is an executable with a fairly simple command-line interface that may generate VHDL source code for most operators described in this book (and a few more).

An operator specification consists of an operator name followed by a list of parameter values.

Hands on: Basic FloPoCo command-line example

The command

```
flopoco IEEEFPAdd wE=8 wF=23
```

produces a file called `flopoco.vhd1` containing synthesizable VHDL for an IEEE754 single precision floating-point adder.

Some operators come as a single VHDL entity, and in other cases, the generated VHDL file contains many entities with a hierarchy of component instantiations. For instance, the previous command outputs the following text on the console, showing four subcomponents:

```
|---Entity RightShifterSticky26_by_max_25_comb_uid4
|   Not pipelined
|---Entity IntAdder_27_comb_uid6
|   Not pipelined
|---Entity LZC_26_comb_uid8
|   Not pipelined
|---Entity LeftShifter27_by_max_26_comb_uid10
|   Not pipelined
|---Entity IntAdder_31_comb_uid13
|   Not pipelined
```

```
Entity IEEEFPAdd_8_23_comb_uid2
Not pipelined
```

The top-level entity, in such cases, is the last one.

It is also possible to have several operator specifications in the same command line.

Hands on: An FPU for Nfloat format

The following command produces an adder, a multiplier, a divider, and a square root operator for the Nfloat equivalent of single precision (all in the same VHDL file).

```
flopoco FPAdd we=8 wf=23 FPMult we=8 wf=23 FPDiv we=8 wf=23
FPSqrt we=8 wf=23
```

Finally, it is possible to specify the value of global parameters such as the frequency in the example below (frequency-directed pipelining is discussed further in Sect. A.2.5). Another important global parameter is target which specifies the target hardware.

Hands on: An FPU pipelined for 200 MHz

The following command pipelines the previous FPU for 200 MHz.

```
flopoco frequency=200 FPAdd we=8 wf=23 FPMult we=8 wf=23
FPDiv we=8 wf=23 FPSqrt we=8 wf=23
```

The size of the generated VHDL code may vary from a few bytes to a few megabytes. It depends on the complexity of the operator, on the value of the parameters, but also on the very nature of the subcomponents: an addition may be described by a single character + in the VHDL implementation, whereas a table of precomputed values, provided in extension, may be arbitrarily large.

A.2.2 More on Parameters

As the above examples show, all the parameters are provided to FloPoCo as *name=value* pairs.

Each operator is heavily parameterized with functional and performance parameters, for reasons described in the introduction of this book (see Sect. 1.5.1, p. 16). To know the parameters of a given operator, just invoke flopoco for this operator without parameters, e.g., flopoco FPAdd.

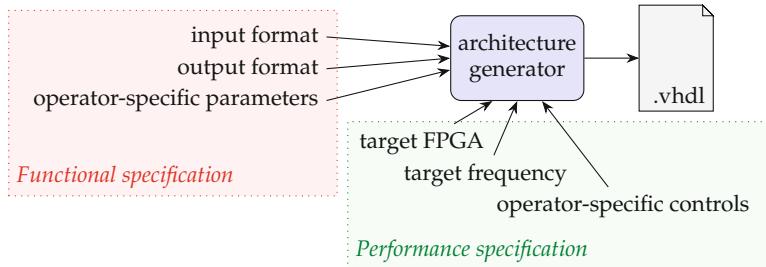


Fig. A.1 Interface to a flopoco Operator

As illustrated in Fig. A.1, parameters can be broadly separated in two classes: *functional* parameters specify the function, while *performance* parameters control the performance and its many trade-offs (such as memory versus compute, area versus speed, frequency versus latency, etc.).

A general rule is that functional parameters should be mandatory with no default value, while performance parameters should be optional with a sensible default value. However, there are exceptions to both rules. For instance, there is one parameter to FPAdd which decides if the operator should be an adder or a subtracter. This is a functional parameter, but its default is set to addition, because we feel it is the most common case. Conversely, FixFunctionByPiecewisePoly has a mandatory parameter for the degree to be used, although it is definitely a performance parameter. The reason here is that we feel that the decision of a “sensible default value” is too application-dependent.

For a given operator, there are usually many parameter combinations that do not make sense. We attempt to filter them and provide sensible error messages in such cases, but users often have more imagination than developers can anticipate, and invalid combinations of parameter values may lead to unexpected crashes or nonworking VHDL.

A.2.3 Do Not Trust FloPoCo, the Test Bench Is Included

FloPoCo is able to generate an infinite number of different operators, and the developers have obviously not tested each of them. This issue is intrinsic to application-specific arithmetic. A simple solution developed in FloPoCo is that each operator comes with a test bench generator that will exercise exactly the operator being generated.

Details about the construction of these test benches can be found in the developer manual (see <http://flopoco.org/> for the latest version). The point to stress here is that the resulting test benches can be trusted by users. The main reason for this is that the test vectors are not built out of the VHDL

being generated, but out of the mathematical specification of the operator (as a mathematical function combined with a well-defined rounding, implemented in C++, see Sect. 1.3, p. 7). Thus, the probability of a bug in the VHDL being hidden by a bug in the test bench is very low. In addition, the code that generates test benches is very small, quite boilerplate, and based on reference multiple-precision libraries that are well specified and well established: GNU Multiple Precision (GMP) and GNU Multiple Precision Floating-point with correct Rounding (MPFR). For this reason, the confidence in the test bench is much higher than the confidence in the generated VHDL.

The test bench generation also provides how the open-source VHDL simulator nvc (<https://github.com/nickg/nvc>) can be invoked to test the circuit. We highly recommend this fast and reliable command-line simulator. The command line is sufficient as it tells the user/developer if everything went fine (0 error(s) encountered) or the test cases that failed. In this case, waveforms are written by nvc, and calls to the open-source waveform viewer gtkwave (<https://gtkwave.sourceforge.net>) are provided that allows to further debug what went wrong.

The user interface to the test bench generation is kept extremely simple, to encourage users to test their FloPoCo operators.

Hands on: Test bench generation in FloPoCo

The following command builds a divider by 3 for 16-bit integers, and an exhaustive test bench for this operator:

```
flopoco IntConstDiv wIn=16 d=3 TestBench
```

This command creates a VHDL file and a (human-readable) `test.input` file containing all the possible values of the input and the corresponding expected output. It also outputs instructions to launch this test using a choice of VHDL simulators.

Hands on: Operation-specific test bench generation

For operators having a large input word size, exhaustive testing is out of reach. In such cases, the test should be limited to a smaller number of test vectors. The following command is an example of a (Nfloat) binary32 floating-point adder with 100,000 random tests.

```
flopoco FPAdd we=8 wf=23 TestBench n=100000
```

This test bench actually begins with all the corner-case test vectors that the developers of FPAdd could think of, including a few regression test vectors corresponding to past bugs.

In addition, the random number generator used there is specific to the operator under test: with two random bit strings as inputs, the prob-

ability of a cancellation in the addition (see Sect. 11.1, p. 331) would be quite low, since it only happens when the exponents differ at most by 1. To properly test this important feature of floating-point addition, the random generator used has been slightly tweaked to make cancellations more frequent. See the developer manual for more details.

A.2.4 Obscure Branches and Code Attics

The installation instructions on the web site currently use the master branch of the Git repository of FloPoCo. Beyond this branch, there are quite a few other, more experimental branches and forks where one may find rarer operators, or code snapshots that ensure that a publication is reproducible.

In principle, developers are encouraged to merge their work in the master branch as soon as it may be of general use. They are also encouraged to clean up the master branch of experimental code, over-parameterization for research purpose, or obsolete variants for which a better option is available. Unfortunately, they often do not have the time for these two time-consuming tasks, and we do apologize for this. Any feedback on what should be prioritized will be welcome.

Quite often, a change to the FloPoCo framework (or to one of the libraries it uses) breaks some operators. If these operators are not fixed by their original developers, the maintainers end up disabling them in the master branch. Usually the code is just unplugged, not removed: it is kept in “attic” directories or “obscure branches,” where it may still compile, or not. In any case, this code is still there for review and/or porting to the current framework, should the need arise. Examples of code which is currently disabled but would deserve to be revived include the HOTBM function approximator by Jérémie Detrey [DD05], the LNS operators by Caroline Collange [VCA10], and the FPGA-specific random number generators by David Thomas [Tho13], among many others.

A.2.5 Automatic Pipelining, the User Point of View

Most operators presented in this book are combinatorial circuits: they have no memory, the result only depends of the input, not of the past history of the operator. This is again a direct consequence of our definition (in Sect. 1.3)

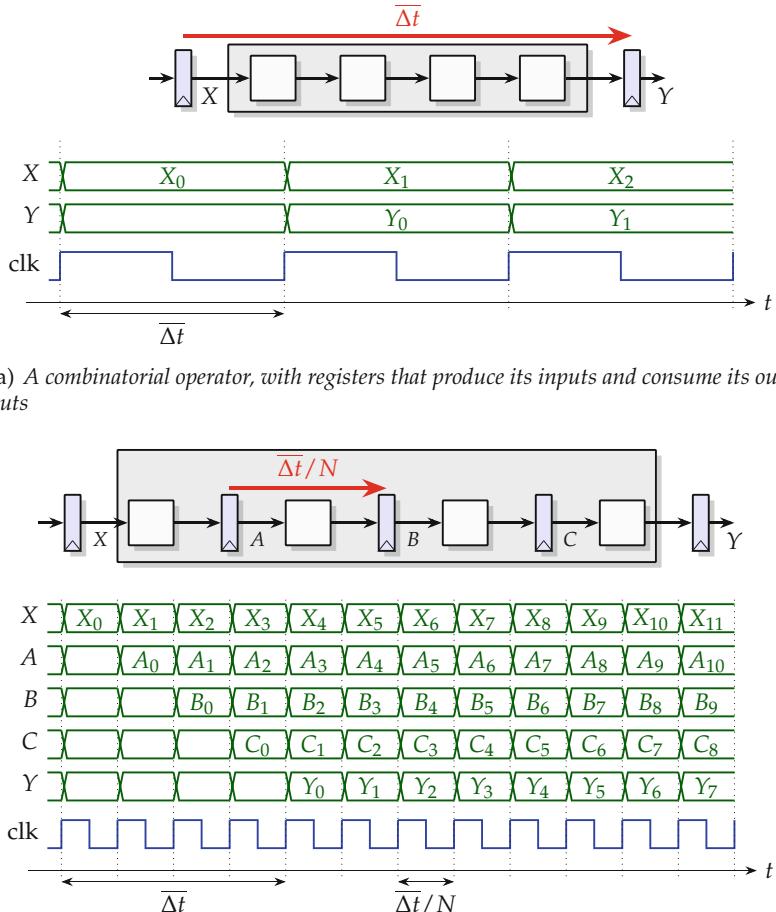


Fig. A.2 Pipelining a combinatorial operator

of an operator as the composition of a mathematical function¹ and a well-defined rounding.

Another point of view is that the directed graph of subcomponents that describes the circuit (down to the gates or whatever elementary processing elements of the technology) is acyclic. However, each component of this graph needs some time (and some energy) to perform its computation. If an input is presented at time t_0 , the correct output will be computed after some

¹ The main exceptions are the floating-point accumulators of Chap. 21 and the filters in Chap. 23. This book also mentions iterative variants of dividers, CORDIC operators, and in general digit-recurrence algorithms, but none of these is currently available in FloPoCo: the tool provides only the unrolled, combinatorial variants.

delay Δt (measured in seconds). This delay may depend on the input. Its maximum value over all the possible inputs is called the *critical path delay* of the circuit, denoted as $\overline{\Delta t}$. If the operator is sandwiched between input registers and output registers as illustrated by Fig. A.2a, then the input registers may present a new input every $\overline{\Delta t}$ period, and the circuit will have the time to process this input (and register the result in the output registers) before the next input is presented. In other words, the operator can operate at frequency $1/\overline{\Delta t}$.

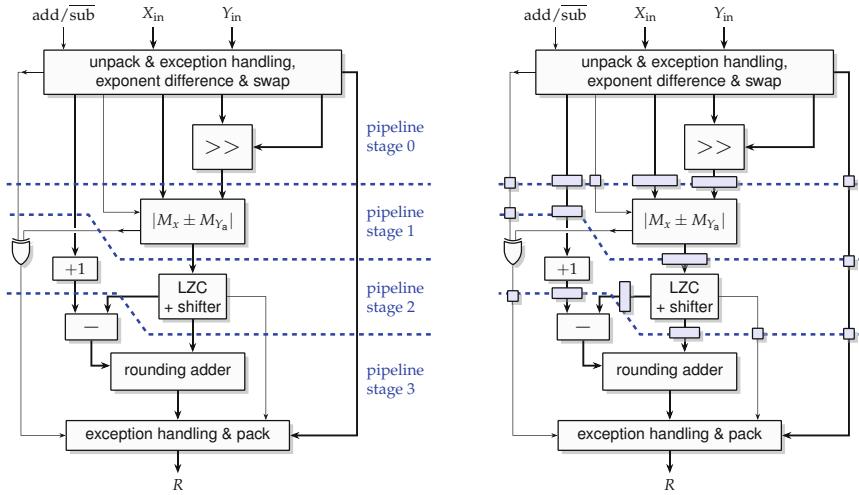
Pipelining is a technique that allows an operator to function at a higher frequency by dividing the circuit graph into N slices called *pipeline stages*, determined such that the critical path delay of each slice is about $\overline{\Delta t}/N$. Registers are inserted between slices to hold intermediate computations, as illustrated by Fig. A.2b. If the pipeline is ideally balanced, and ignoring the delay added by these extra registers, the frequency can become $N/\overline{\Delta t}$: compared to the combinatorial circuit, the frequency has been multiplied by N . The cost is additional registers. In practice, however, it is difficult to achieve a perfectly balanced pipeline, and the registers add some delay which limits the achievable frequency [Hri+02; SC02].

Pipelining combinatorial operators is simple in theory but time-consuming if performed by hand. As Fig. A.3 illustrates, actual operators are often more complex than Fig. A.2. FloPoCo therefore automates this task. The interface for it is simply the target frequency provided through the `frequency` parameter. An important remark on Fig. A.3 is that FloPoCo does not add any input and output registers. All it does is insert the intermediate (pipeline) registers. The “pipeline depth” reports the number of synchronization barriers inserted (this information is also provided in comments before each entity declaration in the generated VHDL).

The resulting pipeline is quite good for practical purposes. Note, however, that FloPoCo does not pretend to generate an optimal pipeline (an optimal pipeline would minimize the overall cost of the registers while achieving a given target frequency [LS91]). It does not even guarantee that the operator will operate at the prescribed frequency – we are at the mercy of the backend tools that will transform the VHDL into an actual circuit. Indeed, these tools are the proper place for fine-tuning a pipeline. However, it does guarantee that the pipeline is well synchronized (and `TestBench` adapts to pipelined operators for users to check). Hence, cases where the required frequency is not reached or even too high can be easily fixed by adjusting the target frequency accordingly.

Hands on: Sandwiching an operator with register

In order to measure the actual operating frequency at which an operator runs, it is important to add the registers on the inputs and outputs. FloPoCo provides an operator that sandwiches between registers the



(a) *Adding 3 synchronization barriers – FloPoCo reports a pipeline depth of 3 – means that there are 4 pipeline stages*

(b) *The corresponding registers are inserted automatically by FloPoCo*

Fig. A.3 Pipelining a floating-point adder

operator preceding it on the command line (just like TestBench tests the preceding operator).

```
flopoco frequency=400 FPAdd wE=8 wF=23 RegisterSandwich
```

FloPoCo also provides (in the `tools/` directory) small python scripts that will synthesize an operator for an FPGA using either Vivado or Quartus. These tools read in the VHDL file the target information, launch synthesis accordingly, and report the main synthesis result on the console.

A.3 FloPoCo for Arithmetic Designers

This section is addressed to prospective developers and intends to provide an overview that helps digging into the code.

A.3.1 Operators and Instances

The core of FloPoCo is the abstract `Operator` class: every VHDL entity produced by FloPoCo corresponds to an `Operator` (i.e., inherits this class). Running FloPoCo constructs a list of `operator` instances (those specified on the command line, and all their subcomponents) and then generates the VHDL for them.

The `Operator` class provides several code generation services. Some are for pure convenience, such as the possibility to declare a signal close to where it is assigned or the management of most of the VHDL boilerplate verbosity. Some are more deeply useful, in particular

- the abstraction of the target hardware described below in Sect. A.3.2,
- the generic framework for operator testing,
- the automatic pipelining framework described in Sects. A.2.5 and A.3.3.

This makes this class quite large, and it is probably not a very good starting point for prospective developers. Instead, they should start with one of the well-documented operators, for instance `FPAAdd`, and lookup `Operator` methods when encountering them.

At the center of the `Operator` class, there is the `vhdl` stream: writing an operator essentially consists in writing VHDL code to this stream, as illustrated by Listings A.1 and A.2.

```

vhdl << declare("signX")           << "<= newX("<<wE+wF<<"); "
vhdl << declare("signY")           << "<= newY("<<wE+wF<<"); "
vhdl << declare(target->logicDelay(), "effSub")  << " <= "
    << " signX xor signY;"        ;
(...)

vhdl << declare(target->adderDelay(wE+1), "eXmeY", wE) << " <= "
    << "(X" << range(wE+wF-1,wF)<<")"
    << " - (Y"<<range(wE+wF-1,wF)<<")"; 
```

Listing A.1 A simple example of VHDL construction (the VHDL code is in blue)

```

signX <= newX(31);
signY <= newY(31);
effSub <= signX xor signY;
(...)
eXmeY <= (X(30 downto 23)) - (Y(30 downto 23)); 
```

Listing A.2 VHDL code produced by the code of Listing A.1

In this example, function `declare()` keeps track of a signal, its word size, and potential delay with respect to the operation performed to produce the signal. It finally outputs the signal name for VHDL code generation. The

delay is used in the automatic pipelining and is target dependent, and the details about that are discussed in the following.

A.3.2 The Target Class Hierarchy

A `Target` is given as argument to the constructor of any `Operator`. A singleton `target` of this class is referenced in `Operator`. `Target` is again an abstract class, which attempts to abstract the features of actual FPGA chips. Classes representing real FPGA chips extend `Target` – we currently have classes for very different FPGAs from Xilinx/AMD and Altera/Intel. Generator code invokes abstract methods declared in `Target`, which are implemented in its subclasses. Thus, the same generator code adapts to all the targets.

The methods provided by the `Target` class can be semantically split into two categories:

- **Architecture-related methods** provide information about the architecture of the FPGA and are used in architectural exploration. For instance, `lutInputs()` returns the number of inputs of the FPGA’s LUTs. This method is used by techniques that are deeply designed around the LUT, for instance, the Ken Chapman’s Multiplier (KCM) [Cha94] of Chap. 12, or the dividers by small integer constants of Chap. 13.
- **Delay-related methods** provide approximative informations about computation time and are essentially used by the pipeline framework. For example, the call `target->logicDelay()` in Listing A.1 returns the logic delay (one LUT) required by the `xor` expression given in the VHDL statement. Another example is the call to `adderDelay(int n)`: it returns the delay of an n -bit addition.² Thus, `adderDelay(16)` will, for instance, return different values for different FPGAs, and eventually the pipeline will be deeper for a slower FPGA.

The reliance on the virtual `Target` class ensures that FloPoCo datapaths are designed in a reasonably future-proof way, as soon as this abstract class is reasonably implemented for each new FPGA model of interest.

Of course, it is also possible for an `Operator` to produce very target-specific code, triggered by an explicit test on the target name.

Capturing embedded memories and DSP blocks is more complex, since these blocks have internal registers, various chaining possibilities, dual porting, etc. These features are best used through FloPoCo operators such as

² This used to be a simple affine function of n , but this method has become more and more complicated as the FPGAs become more heterogeneous: LUTs are grouped in slices or logic array blocks (LABs) – see Chap. 4 – and there are different carry propagation delays inside such a group or between groups. However, it remains possible to have a fairly accurate timing model of a complete addition for any size n as long as it does not exceed the diameter of the FPGA.

Table and `IntMultiplier`, which take care of this complexity. Inside these operators, we are not ashamed to have target- or vendor-specific ad hoc implementations.

A.3.3 Automatic Pipelining

In principle, pipeline construction could be fully automatic in synthesis tools: the timing information could be extracted from the generated circuit and used to build a balanced pipeline. It is even possible to determine the optimal placement of pipeline registers to minimize their overall cost [LS91]. FPGA and ASIC synthesis tools for VHDL synthesis have limited support for pipeline balancing. HLS tools for FPGAs are progressing toward this goal, but the pipelines they produce are often much deeper than what a more manual approach can achieve.

FloPoCo strives to provide a close estimate of the actual achievable pipeline depth. Unfortunately, FloPoCo ignores the semantics of what designers place in the `vhdl` stream, and recovering this information accurately would require a complete VHDL compiler. Therefore, the current approach is to burden the designers with the task of modeling the timing of their VHDL.

The good news, however, is that this timing information can be expressed in a generic way, so that FloPoCo is then able to pipeline the operator for arbitrary frequencies (within sensible limits) and for any target hardware.

In practice, operator design now consists of two well-separated tasks:

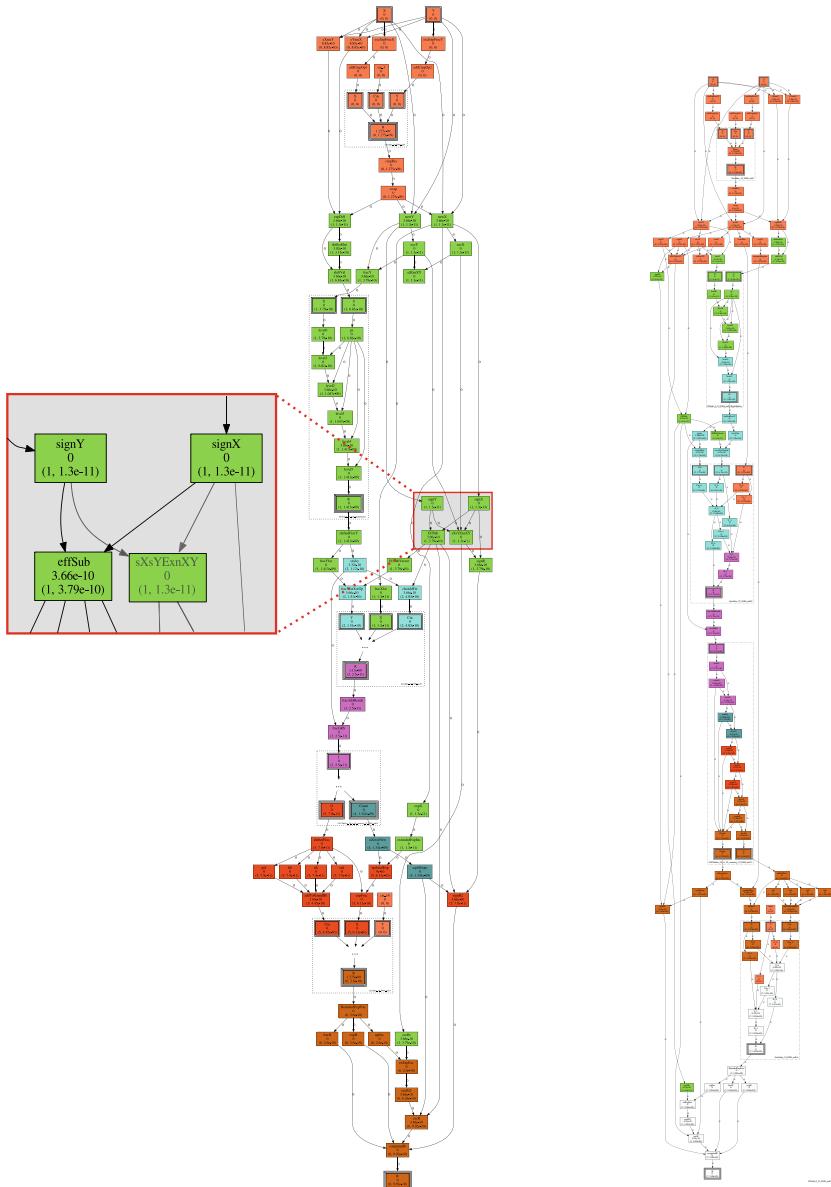
1. building a functional combinatorial datapath,
2. adding timing information to this datapath so that it can be pipelined.

In the first task, the main concern should be that the operator is correct (it passes the `TestBench`). Performance will also be a concern, but with a simple metrics: the critical path delay of the combinatorial operator (measured register-to-register, thanks to the `RegisterSandwich` operator).

The second tasks consists in adding, to the C++ code, invocations of `target` methods that model the timing of the generated VHDL. Examples are the `target->logicDelay()` and `target->adderDelay()` calls in Listing A.1. This second task will not change the function computed by the operator (and the designer does not have to modify any of the code placed in the `vhdl` stream). The worst that may happen is that a poor timing model will lead to an inefficient pipeline (the target frequency is not reached, or is reached with more stages than needed).

Once this timing information is added, pipelining consists of three passes [ID17]:

1. FloPoCo parses the `vhdl` stream to detect signal names, builds a dependency graph of these signal names, and labels this graph using the



(a) *Graph for Virtex6, with the instances as black boxes.* (b) *Graph for StratixV, with the instances flattened.*
The zoom shows the part corresponding to Listing A.1.

Fig. A.4 Some dependency graphs generated by FloPoCo for FPAAdd(8, 23) pipelined for 400 MHz. Each color corresponds to a pipeline stage

designer-provided timing information. For instance, in Listing A.1, one edge of the dependency graph will capture that signal `exmeY` depends on signals `x` and `y`, and more precisely that it will only be available δt seconds after both `x` and `y` are available, where δt is the delay of an addition of $w_E + 1$ bits on this target.

2. Then, FloPoCo schedules this graph, i.e., assigns a cycle to each signal. The current approach is an heuristic that schedules computations as soon as possible.³
3. Finally, the actual VHDL is generated from the scheduled graph and the `vhdl` stream. This simply consists in delaying some signals and replacing signal names on the right-hand side of `<=` with their delayed version: in the final VHDL, a signal `effSub_d3` should be read as “signal `effSub`, delayed by 3 cycles”.

For the interested reader, this technique is fully detailed in [ID17]. The pipelining framework also generates dependency graphs (by setting the global parameter `dependencyGraph` to `compact` or `full`) annotated with timing information, some examples of which are shown in Fig. A.4.

Circuit time

In a combinatorial circuit, the timing of a signal s can be computed by accumulating the delays on the longest simple path from the earliest input to s (critical path delay).

Inside a pipelined circuit, however, a signal s may be delayed by a number of cycles c . The timing of s is then expressed as a pair (c, τ) , where

- c is an integer that counts the number of registers on the longest path from an input to s .
- τ is a real number that represents the critical path delay (in seconds) from the last register or earliest input to s .

There is a lexicographic order on such timings:

$$(c_1, \tau_1) > (c_2, \tau_2) \text{ iff } c_1 > c_2 \text{ or } (c_1 = c_2 \text{ and } \tau_1 > \tau_2).$$

The task of the FloPoCo scheduler is to associate a lexicographic time to each signal.

³ This approach does ensure the minimal number of cycles, but not the minimal number of registers. Leiserson and Saxe optimal retiming [LS91] can typically reduce the register cost by 20–30% [Par22] but is not implemented at the time of writing this book.

A.3.4 The BitHeap Framework

FloPoCo also provides extensive support for the bit heaps introduced in Chap. 7. Beware that the concept of bit heap is orthogonal to the concept of operator (or VHDL entity).

- It is common that large operator involve several bit heaps (see, for instance, Fig. 7.10, p. 159).
- It is also sometimes useful to have a bit heap shared by several operator instances. A canonical example is that of a sum of products by constants (SOPC) architecture presented in Sect. 12.5.5, p. 412. An efficient implementation requires that each constant multiplier adds bits to a common bit heap but does not perform its compression. In such cases, the implementation of the constant multiplier does not correspond to a VHDL entity: there is a method that inputs a pointer to another operator and outputs code to the vhdl stream of this operator.

A.3.4.1 Interface Overview

The main interface to the developer is the `BitHeap` C++ class, which encapsulates all the needed data structures and methods. The bits to be added to a `BitHeap` are simply signals of the operator being generated. The initial bit heap thus defined may be of any shape or size. The `BitHeap` class also implements the compressor tree optimization techniques presented in Sect. 7.3 (the compression algorithm is selected by the global parameter `compression`) and can generate the corresponding hardware.

The typical code to generate the architecture of an operator involving a bit heap is shown in Listing A.3. The main methods to manipulate the bit heap are shown in Table A.1. Basically, there is everything to add or subtract single bits, signed or unsigned numbers held in bit vectors, or constants. Once all the bits have been thrown on the bit heap, the `startCompression()` method launches the computation of the compressor tree as well as the VHDL code generation.

A.3.4.2 The Data Structure

This section and the following describe the internals of the `BitHeap` class and are intended for developers wishing to extend the `BitHeap` framework itself (e.g., with new compressors or compression algorithms).

A *weighted bit* is a data structure consisting essentially of a signal name, a bit position, and various fields used when building the compressor tree. In FloPoCo, the `Signal` class also encapsulates the timing of each signal, so each weighted bit also carries its arrival time. This timing information is

```

bh = new BitHeap(...); // create an empty bit heap

// construction of the bit heap
for (...) {
    // generate VHDL that defines signals such as
    /// mySingleBit or myBitVector
    ...

    // then add these signals to the bit heap:
    bh->addBit(myBit, pos1); // no VHDL generation here
    bh->addSignal(mybitVector, pos2); // nor here
}

// generate the compressor tree for the bit heap
bh->startCompression(); // this line generates a lot of VHDL

```

Listing A.3 Typical code for an operator involving a bit heap**Table A.1** The main methods of the BitHeap interface

Method	Description
void addBit(string sigName, int pos)	Add a single bit
void addSignal(string sigName, int shift)	Add a fixed-point signal, with sign extension if needed
void subtractSignal(string name, int shift)	Subtract a fixed-point signal, with sign extension if needed
void addConstantOneBit(int pos)	Add a constant one bit
void addConstant(mpz_class cst, int pos)	Add the constant cst · 2 ^{pos}
void startCompression()	Generate a compressor tree

When the argument is a signal name, it is used to refer to a FloPoCo object of the class Signal, which encapsulate a fixed-point format with its signedness, its MSB, and its LSB

represented using the lexicographic circuit time of pipelined circuits introduced in Sect. A.3.3.

A column of the bit heap is represented as a list of weighted bits. This list is sorted by the arrival time of the bits, so that compression algorithms can compress first the bits arrived first.

The complete bit heap data structure essentially consists of its MSB and LSB and an array of columns indexed by the positions.

The BitHeap class also offers methods to visualize a bit heap as a dot diagram like the one shown in Fig. 7.18, p. 173.

FloPoCo may generate such figures as Scalable Vector Graphics (SVG) files that can be opened in a browser, in which case hovering the mouse over

one bit shows its signal name and its arrival instant in circuit time. Different arrival times are there indicated by different colors.

A.3.4.3 Compressor Tree Generation

Once the data structure of the BitHeap is created by using the methods of Table A.1, the compressor tree generation is started by calling `startCompression()`. It involves the following steps:

1. The algorithm for solving the compressor tree problem is selected (a derived class from `CompressionStrategy`).
2. A list of possible (target-dependent) compressors is generated (class `BasicCompressor` for representing the shape and class `Compressor` generating the hardware performing the compression).
3. Then, the bits of the bit heap are scheduled to different stages according to their cycle and combinatorial arrival time.
4. Next, the compressor trees are optimized (by a class derived from `CompressionStrategy`), and a `BitHeapSolution` object is created. This solution contains the used compressors per stage and column.
5. Finally, the VHDL code of the compressor tree is generated.

New compressor tree optimization methods can be implemented by extending `CompressionStrategy`.

References

- [Bru+13] Nicolas Brunie, Florent de Dinechin, Matei Istoan, Guillaume Sergent, Kinga Illyes, and Bogdan Popa. “Arithmetic Core Generation Using Bit Heaps”. In: *Field Programmable Logic and Application (FPL)*. IEEE, 2013, pp. 1–8. (cit. on p. 761).
- [CDP22] Maxime Christ, Florent de Dinechin, and Frédéric Pétrot. “Low-precision logarithmic arithmetic for neural network accelerators”. In: *International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE, 2022. (cit. on p. 761).
- [Cha94] Ken D. Chapman. “Fast Integer Multipliers Fit in FPGAs”. In: *Electronic Design News* (1994). (cit. on pp. 15, 771).
- [DD05] Jérémie Detrey and Florent de Dinechin. “Table-based polynomials for fast hardware function evaluation”. In: *International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE, 2005, pp. 328–333. (cit. on p. 766).
- [DI15] Florent de Dinechin and Matei Istoan. “Hardware implementations of fixed-point Atan2”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2015, pp. 34–41. (cit. on pp. 307, 316, 761).

- [Din+13] Florent de Dinechin, Pedro Echeverría, Marisa López-Vallejo, and Bogdan Pasca. "Floating-Point Exponentiation Units for Reconfigurable Computing". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 6.1 (2013). (cit. on p. 761).
- [Din12] Florent de Dinechin. "Multiplication by Rational Constants". In: *IEEE Transactions on Circuits and Systems, II* 59.2 (2012), pp. 98–102. (cit. on p. 761).
- [DIS13] Florent de Dinechin, Matei Istoan, and Guillaume Sergent. "Fixed-Point Trigonometric Functions on FPGAs". In: *SIGARCH Computer Architecture News* 41.5 (2013), pp. 83–88. (cit. on p. 761).
- [DJP10] Florent de Dinechin, Mioara Joldes, and Bogdan Pasca. "Automatic generation of polynomial-based hardware architectures for function evaluation". In: *Application-specific Systems, Architectures and Processors*. IEEE, 2010. (cit. on p. 761).
- [DKP09] Florent de Dinechin, Cristian Klein, and Bogdan Pasca. "Generating High-Performance Custom Floating-Point Pipelines". In: *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2009, pp. 59–64. (cit. on p. 761).
- [DP10] Florent de Dinechin and Bogdan Pasca. "Floating-point exponential functions for DSP-enabled FPGAs". In: *Field Programmable Technologies*. Best paper candidate. 2010, pp. 110–117. (cit. on p. 761).
- [FSZ22] Nicolai Fiege, Patrick Sittel, and Peter Zipf. "Optimal Binding and Port Assignment for Loop Pipelining in High-Level Synthesis". In: *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2022, pp. 262–269. (cit. on p. 762).
- [GMK19] Mario Garrido, Konrad Möller, and Martin Kumm. "World's Fastest FFT Architectures: Breaking the Barrier of 100 GS/s". In: *Transactions on Circuits and Systems I: Regular Papers* 66.4 (2019), pp. 1507–1516. (cit. on p. 761).
- [Hab+22] Tobias Habermann, Jonas Kühle, Martin Kumm, and Anastasia Volkova. "Hardware-Aware Quantization for Multiplier-less Neural Network Controllers". In: *Asia Pacific Conference on Circuits and Systems (APCCAS)*. IEEE, 2022, pp. 1–5. (cit. on p. 761).
- [Har+22] Martin Hardieck, Nicolai Fiege, Martin Kumm, and Peter Zipf. *Elysia – Project Website*. 2022. URL: <http://uni-kassel.de/go/elysia>. (cit. on p. 762).
- [Hri+02] M. S. Hrishikesh, Doug Burger, Norman P. Jouppi, Stephen W. Keckler, Keith I. Farkas, and Premkishore Shivakumar. "The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays". In: *29th annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2002, pp. 14–24. (cit. on p. 768).
- [ID17] Matei Istoan and Florent de Dinechin. "Automating the pipeline of arithmetic datapaths". In: *Design, Automation and Test in Europe (DATE)*. IEEE, 2017. (cit. on pp. 761, 772, 774).

- [KLZ12] Martin Kumm, Katharina Liebisch, and Peter Zipf. "Reduced Complexity Single and Multiple Constant Multiplication in Floating Point Precision". In: *International Conference on Field Programmable Logic and Application (FPL)*. IEEE, 2012, pp. 255–261. (cit. on p. 761).
- [Kum+16] Martin Kumm, Oscar Gustafsson, Mario Garrido, and Peter Zipf. "Optimal Single Constant Multiplication using Ternary Adders". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 65.7 (2016), pp. 928–932. (cit. on p. 761).
- [Kum+22] Martin Kumm, Konrad Möller, Patrick Sittel, and Peter Zipf. *Origami HLS – Project Website*. 2022. URL: <http://www.uni-kassel.de/go/origami>. (cit. on p. 762).
- [LS91] C. E. Leiserson and J. B. Saxe. "Retiming Synchronous Circuitry". In: *Algorithmica* 6.1 (1991), pp. 5–35. (cit. on pp. 768, 772, 774).
- [MDB20] Raul Murillo, Alberto A. Del Barrio, and Guillermo Botella. "Customized Posit Adders and Multipliers using the FloPoCo Core Generator". In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2020. (cit. on p. 761).
- [NPP11] Hong Diep Nguyen, Bogdan Pasca, and Thomas B. Preußen. "FPGA-Specific Arithmetic Optimizations of Short-Latency Adders". In: *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2011, pp. 232–237. (cit. on p. 761).
- [Par22] Rémi Parrot. "Réseaux de Petri temporisés pour la synthèse de circuits pipelinés". PhD thesis. École Centrale de Nantes, 2022. (cit. on p. 774).
- [SC02] Eric Sprangle and Doug Carmean. "Increasing processor performance by implementing deeper pipelines". In: *29th annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2002, pp. 25–34. (cit. on p. 768).
- [Sit+17a] P Sittel, M Kumm, K Möller, M Hardieck, and P Zipf. "High-Level Synthesis for Model-Based Design with Automatic Folding including Combined Common Subcircuits". In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*. 2017, pp. 103–113. (cit. on p. 762).
- [Sit+17b] Patrick Sittel, Konrad Möller, Martin Kumm, Peter Zipf, Bogdan Pasca, and Mark Jervis. "Model-Based Hardware Design Based on Compatible Sets of Isomorphic Subgraphs". In: *International Conference on Field-Programmable Technology (FPT)*. IEEE, 2017, pp. 199–202. (cit. on p. 762).
- [Sit+19] Patrick Sittel, Nicolai Fiege, Martin Kumm, and Peter Zipf. "Isomorphic Subgraph-based Problem Reduction for Resource Minimal Modulo Scheduling". In: *Reconfigurable Computing: Architectures, Tools and Applications. International Conference on Re-*

- configurable Computing and FPGAs (ReConFig). 2019, pp. 1–8. (cit. on p. 762).
- [Tho13] David B. Thomas. “Parallel generation of Gaussian random numbers using the Table-Hadamard transform”. In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2013. (cit. on pp. 761, 766).
- [Tho15] David B. Thomas. “A general-purpose method for faithfully rounded floating-point function approximation in FPGAs”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2015. (cit. on p. 761).
- [Uğ+17] H. Fatih Uğurdağ, Florent de Dinechin, Y. Serhan Gener, Sezer Gören, and Laurent-Stéphane Didier. “Hardware division by small integer constants”. In: *IEEE Transactions on Computers* 66.12 (2017), pp. 2097–2110. (cit. on p. 761).
- [VCA10] Panagiotis D. Vouzis, Caroline Collange, and Mark G. Arnold. “A Novel Cotransformation for LNS Subtraction”. In: *Journal of Signal Processing Systems* 58.1 (2010), pp. 29–40. (cit. on pp. 761, 766).
- [Vol+19] Anastasia Volkova, Matei Istoan, Florent de Dinechin, and Thibault Hilaire. “Towards Hardware IIR Filters Computing Just Right: Direct Form I Case Study”. In: *IEEE Transactions on Computers* 68.4 (2019). (cit. on p. 761).

APPENDIX B

Optimization Using Integer Linear Programming

Integer linear programming (ILP) is a family of optimization techniques for which very powerful solvers exist. The purpose of this appendix is to provide arithmetic designers with the necessary background for formalizing and solving arithmetic problems using this powerful tool.

Application-specific arithmetic involves a lot of combinatorial optimization problems. Sometimes the trade-offs are trivial to investigate, sometimes they are small enough to be solved by exhaustive enumeration, but sometimes tricky problems appear, including NP-hard optimization problems. Many such NP-hard problems can be described in the formalism of integer linear programming (ILP). For illustration, ILP is used in this book for compressor trees (Sect. 7.3.3.2, p. 189), multiplication (Sect. 8.4.3, p. 233), constant multiplication (Sect. 12.1.4, p. 373), function approximation (Sect. 17.2.11, p. 522 and Sect. 18.4.4, p. 566), as well as digital filters (Chap. 23, p. 669).

The motivation for expressing a problem using ILP is to focus on describing the optimization problem and not on writing an algorithm that will solve it: for this, an existing ILP solver may be used. Benefiting from decades of research, modern ILP solvers are very powerful and can address very large problems. ILP is not the only generic problem-solving technique: SAT solving and constraint programming are also extremely powerful and would probably deserve a similar appendix. We focus on ILP because it has proven well suited to arithmetic problems.

This appendix does not cover the techniques used by ILP solvers, and the interested reader is referred to textbooks on this topic [Sch98; SZ15]. The purpose here is to give a practical tutorial on formulating arithmetic optimization problems using ILP.

In the following, Sect. B.1 introduces linear programming (LP) and its variants using integer and binary variables. Section B.2 is a tutorial on using

solvers with two common input languages (LP, AMPL/GNU MathProg). Section B.3 addresses practical issues in problem modeling, in particular common patterns from the models introduced throughout the book. Finally, limitations of current ILP solvers are discussed in Sect. B.4.

B.1 Linear Programming Problems

Linear programming is a special case of mathematical programming. Here, the term *programming* is used in a historical context and does not refer to computer programming but more to optimization.

Here is the generic formulation of a linear programming problem:

$$\begin{array}{ll} \text{minimize} & \mathbf{c}^T \mathbf{x} \\ \text{subject to} & \mathbf{Ax} \leq \mathbf{b}, \\ & \mathbf{x} \geq \mathbf{0}, \text{ and } \mathbf{x} \in \mathbb{R}^n, \end{array}$$

where

- \mathbf{x} is a vector of n positive real variables,
- \mathbf{c} is a vector defining the cost of each variable,
- matrix \mathbf{A} and vector \mathbf{b} define the constraints of the problem.

The number of constraints is independent of the number of variables. The objective can be turned into a maximization problem mathematically by inverting the signs of \mathbf{c} . Practically, we just write *maximize* instead of *minimize* in this case. The term $\mathbf{c}^T \mathbf{x}$ is called the *objective function*, while $\mathbf{c}^T \mathbf{x}'$ for any specific value of \mathbf{x}' is called the *objective value*.

Constraints requiring \geq relations can be obtained by simply negating both sides of the constraint. Constraints requiring $=$ relations can be obtained by \geq and \leq , i.e., $x_1 = 5$ can be obtained by $5 \leq x_1 \leq 5$. Hence, the relations \geq , \leq , and $=$ can be used for practical linear program descriptions and are directly supported by solvers. Note that the strictly less/greater relations ($>$, $<$) are not supported and have to be treated differently (e.g., replacing $x > c$ by $x \geq c + \delta$ for some small enough δ). Actually, solvers come with input languages for expressing linear programming problems which are much more user-friendly than the matrix formulation above. However, it is important to understand from the start that the expressive power of these languages remains limited to linear problems that can be reduced to this simple matrix formulation.

B.1.1 A First Example Problem

Before going into more details, let us start with a first example using the definitions above:

$$\begin{aligned} & \text{maximize} && \begin{pmatrix} 2 \\ 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \\ & \text{subject to} && \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \leq \begin{pmatrix} 6 \\ 1 \end{pmatrix}, \\ & && \mathbf{x} \geq \mathbf{0}, \text{ and } \mathbf{x} \in \mathbb{R}^2. \end{aligned}$$

We can rewrite this into a bit more readable explicit form, resulting in:

$$\begin{aligned} & \text{maximize} && 2x_1 + 3x_2 \\ & \text{subject to} && x_1 + x_2 \leq 6, \\ & && x_2 - x_1 \leq 1, \\ & && x_1, x_2 \geq 0, \text{ and } x_1, x_2 \in \mathbb{R}^2, \end{aligned}$$

Clearly, without any constraint, the optimal solution would be unbounded. The problem gets more interesting with the introduction of constraints that bound our variables. The reader should think about which values of x_1 and x_2 fulfill the constraints and reach a maximum in the objective.

In this two-dimensional problem, the solution space can be graphically represented on the plane as given in Fig. B.1. Each constraint corresponds to a line separating the feasible region (where the constraint is fulfilled) from the unfeasible region (where it is not). The overall feasible region (where all constraints are fulfilled) is shown in green. The optimal values of x_1 and x_2 are found at the intersection of the two lines (point “optimal LP”) which equals $x_1 = 2.5$ and $x_2 = 3.5$. The reader may check that these values just fulfill all constraints. The objective value is then $2 \times 2.5 + 3 \times 3.5 = 15.5$.

B.1.2 Integer, Binary, and Mixed-Integer Linear Programming

By changing the domain of the variables to integer numbers (i.e., $\mathbf{x} \in \mathbb{N}^n$) the problem is called integer linear programming (ILP). This also heavily changes the complexity of the problem in practice. This may sound counter-intuitive as the infinite solution space when $\mathbf{x} \in \mathbb{R}^n$ becomes finite in ILP. The optimal solution to our example problem above when $x_1, x_2 \in \mathbb{N}^2$ is $x_1 = 3$ and $x_2 = 3$. This solution is illustrated by the point “optimal ILP” in Fig. B.1.

A practically important special case happens when all variables are binary: then the problem is called binary integer linear programming (BILP)

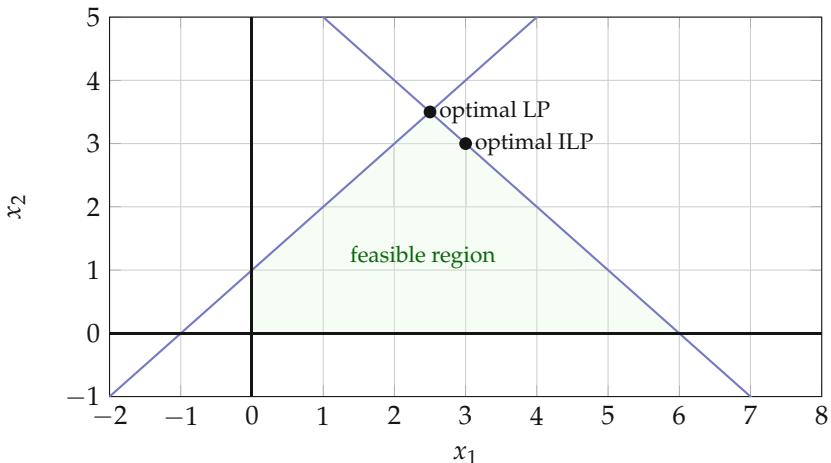


Fig. B.1 Example showing the feasible region as well as the optimal solution for the example problem

or 0–1 ILP. The binary variables are often called decision variables as they decide between two binary options.

Finally, a problem mixing real, integer, and binary variables is called a mixed integer linear programming (MILP) problem. It is common in application-specific arithmetic to have constraints between bits, integers, (representing fixed-point values), and reals (representing the values that an operator must approximate).

B.2 A Tutorial on Using ILP Solvers

There are many good solvers available that can solve MILP problems. Examples are the commercial solvers Gurobi [[Gurobi](#)] and CPLEX [[IBM22](#)] as well as open-source solvers like SCIP [[Ach09; ZIB22](#)], lp_solve [[Tec22](#)] or the GNU Linear Programming Kit (GLPK) [[Inc22a](#)]. These tools are routinely able to solve problems with thousands of variables and thousands of constraints.

They are also fast enough to enable the following strategy that is often used for nonlinear problems. The parameters that make the problem nonlinear are identified, and the problem is decomposed into subproblems where these parameters are considered as constant. These subproblems are now linear, and ILP solvers may be invoked on them. An external loop enumerates the subproblems and keeps the optimal solution.

When invoking an ILP solver on a complex problem, several situations may occur. The tool may answer that no feasible solution exists, which ac-

```

maximize
  obj: 2 x1 + 3 x2
subject to
  c1: x1 + x2 <= 6
  c2: x2 - x1 <= 1
bounds
  x1 >= 0
  x2 >= 0
general
  x1 x2
end

```

Listing B.1 Example in LP file format

tually means that there is no solution to the problem as described. Or, it may provide a solution, which is then the optimal solution to the problem as described. A third situation is that the tool takes too long to answer. In such cases, it is possible to set a timeout after which the tool will return the best feasible solution found so far. In this case, this solution fulfills all the constraints (otherwise the tool reports that no solution was found), but it is not guaranteed to be an optimal one. Still, it is very often good enough for practical purposes.

A common pitfall of using ILP solvers is to model only a subset of the possible solutions. This means that the optimal found by the tool is not the optimal of the underlying problem.

B.2.1 Using Stand-Alone Tools

An ILP solver can be used as a stand-alone program that inputs problem descriptions in some specific language. Among these, the LP file format is a human-readable description language supported by most solvers. Listing B.1 shows the LP file for the example above (including integer variables, declared within the general statement). For instance, the Gurobi tool can be invoked with the following command line, where `example.lp` is a file containing Listing B.1:

```
gurobi_cl ResultFile=example.sol example.lp
```

This results in a solution file `example.sol` with the following content:

```

# Solution for model obj
# Objective value = 15
x1 3
x2 3

```

```

var x1 >= 0, integer;
var x2 >= 0, integer;

maximize z:      2*x1 + 3*x2;

subject to c11:   x1 +   x2 <= 6;
subject to c12:   x2 -   x1 <= 1;

end;

```

Listing B.2 Example in GNU MathProg format

```

param N_A := 5; # the number of adders

set adders := {1..N_A}; # the set of adders
set inputs := {'left','right'}; # the set of adder-inputs

#true when input 'i' of adder 'a' is connected to adder 'k'
var c{adders,inputs,adders} >= 0, binary;

subject to c3b {a in adders,i in inputs}:
    sum{k in adders} c[a,i,k] = 1;

end;

```

Listing B.3 Example of the more complex constraint C3b

One drawback of the LP file format is that each constraint and each variable has to be explicitly defined. Practical problems often involve sets of similar constraints that, for example, differ only by some index (see, e.g., the ILP formulation of compressor tree synthesis in Sect. 7.3.3, p. 190).

The algebraic modeling language AMPL (A Mathematical Programming Language) provides a much more compact way of modeling such cases. The AMPL translator is proprietary software, but the GNU MathProg Modeling Language [MS20], which is part of GLPK, supports a subset of AMPL. Listing B.2 shows the GNU MathProg description of our example. GNU MathProg is much more powerful than this simple example shows. For example, it allows to define sets, multidimensional arrays of variables, sum expressions, etc. The interested reader is referred to the introduction given in [SZ15, Appendix F] or the manual [MS20].

As example, consider constraint C3b for the MCM problem of Sect. 12.1.4.2, p. 375,

$$\mathcal{C}3b : \quad \sum_{k=1}^{a-1} c_{a,i,k} = 1 \quad \forall a = 1 \dots N_A, i \in \{L, R\}.$$

Listing B.3 shows its GNU MathProg description for $N_A = 5$ adders. The same model as LP file would require an explicit line for each adder and input and an explicit sum.

There is also a nice website to model and solve GNU MathProg programs online available at <https://online-optimizer.appspot.com>. It is based on GLPK and was developed by the authors of [SZ15].

B.2.2 ILP Solvers Embedded Within Other Scripting Language

Most solvers provide a library that can be linked to an application that requires the optimizer. This enables a very convenient way to experiment with LP problems through scripting languages such as Python or Julia. Such environments provide an interface to invoke a solver's library, along with all the other libraries familiar to programmers.

B.2.3 ScaLP, a Common C++ Interface to ILP Solvers

The interfaces provided by the solver libraries of the vendors are unfortunately quite different. The solvers also have different strengths and weaknesses; therefore, it is useful to be able to link to several of them. This is addressed by wrapper libraries like OR-Tools [Inc22b] or ScaLP, a Scalable Linear Programming Library [SSK22; Sit+18], which is used within FloPoCo. ScaLP provides a lightweight C++ library that dynamically links different solvers at runtime. It also uses operator overloading to make the C++ formulation of linear programming problem as readable as possible.

Listing B.4 shows the C++ code of the linear programming problem of our running example. A `Solver` object is created using the `wishlist` constructor, i.e., it is searched for the Gurobi solver, if this is not available, it is searched for CPLEX, etc. Once a solver is found, the problem is created and solved, and the output is printed. Other methods allow to access the result provided by the solver for further processing.

B.3 Practical Problem Modeling with ILP

There is no systematic way to formulate a combinatorial optimization problem as a good ILP description: this often remains an art. The following provides a collection of practical modeling techniques that share some common patterns in the ILP formulations used in this book. This is by far not complete, and the reader may refer to textbooks that consider modeling [Wil13].

```

ScaLP::Solver s =
  ScaLP::Solver(
    ScaLP::Solver({ "Gurobi", "CPLEX", "SCIP", "LPSolve" })
  );

ScaLP::Variable x1 = ScaLP::newIntegerVariable("x1");
ScaLP::Variable x2 = ScaLP::newIntegerVariable("x2");

// Set the Objective
s.setObjective(ScaLP::maximize(2*x1 + 3*x2));

// add the Constraints
s << (x1 + x2 <= 6);
s << (x2 - x1 <= 1);

// Try to solve
ScaLP::status stat = s.solve();

// print results
std::cout << "The result is " << stat << std::endl;
std::cout << s.getResult() << std::endl;

```

Listing B.4 Example ILP implemented in C++ using ScaLP

B.3.1 Using Boolean Variables to Model Decision Problems

Many of the problems addressed in this book are variants of *decision problems*, where Boolean variables $\mathbf{x} \in \{0,1\}^N$ (the set of *decision variables*) are used to encode whether or not a certain decision is made. Decision variables are used, for instance, to select among alternative implementation options within an operator. For example, a decision variable can define if a certain compressor in a certain stage and column in a compressor tree is used or not. Another example is a set of decision variables that define which word size is selected in a particular place of an operator. Each decision can be linked with a certain cost in the objective, and constraints can define valid combinations of decision variables. For example, a constraint could make sure that the overall output error related to some word size is sufficiently small.

B.3.2 Translating Boolean Relations into Constraints

Decisions are often not independent. Hence, there are often some logical relations between Boolean variables that can be modeled by using constraints. An example is that only one out of mutually exclusive alternatives should be selected.

Table B.1 Boolean expression and how to model them using linear expression(s)

Boolean expression	Analytic expression	Linear expression(s)
\bar{x}	$1 - x$	$1 - x$
$x \wedge y = 1$	$xy = 1$	$x = 1$ $y = 1$
$x \vee y = 1$	$x + y \geq 1$	$x + y \geq 1$
$x \oplus y = 1$	$x + y = 1$	$x + y = 1$
		$z \leq x$
$x \wedge y = z$	$xy = z$	$z \leq y$ $z + 1 \geq x + y$
		$z \geq x$
$x \vee y = z$	$x + y - xy = z$	$z \geq y$ $z \leq x + y$
$x \oplus y = z$	$x + y - 2xy = z$	$z = x + y - 2w$ with $w \in \mathbb{Z}$
$x \rightarrow y = \bar{x} \vee y$	$1 - x + xy$	$x \leq y$

The following introduces linear algebraic expressions for the Boolean relations AND (\wedge), OR (\vee), and NOT (\neg), using Boolean variables $w, x, y, z \in \{0, 1\}$. The corresponding rules are summarized in Table B.1.

NOT

The negation of a variable can be simply expressed algebraically as

$$\bar{x} = 1 - x. \quad (\text{B.1})$$

This is a linear expression and can be applied in ILP as is.

AND

A logical AND can be expressed algebraically as a multiplication

$$x \wedge y = xy, \quad (\text{B.2})$$

but the term xy is not a linear combination of x and y . The constraint

$$x \wedge y = 1, \quad (\text{B.3})$$

is easy to express as the two linear constraints

$$x = 1 \quad (B.4)$$

$$y = 1. \quad (B.5)$$

This may seem a bit academic, as x and y are then both already decided, but it is of practical use when x and y are defined as more complex expressions.

It is more tricky to define a variable as the AND of two other variables:

$$z = x \wedge y = xy. \quad (B.6)$$

The nonlinear term xy can be expressed as the following linear constraints:

$$z \leq x \quad (B.7)$$

$$z \leq y \quad (B.8)$$

$$x + y \leq z + 1. \quad (B.9)$$

The reader may check by evaluating the four different combinations of x and y that the linear constraints are valid if and only if the Boolean AND is true.

As any Boolean relation can be expressed by AND and NOT, this is actually enough to model any Boolean relation. However, it is not always the best way to do so. More elegant ways, using fewer variables, can often be found for specific cases. Hence, we continue with the OR relation.

OR

A logical OR can be expressed analytically as

$$x \vee y = x + y - xy. \quad (B.10)$$

To express the constraint that x or y (or both) should be true, or

$$x \vee y = 1, \quad (B.11)$$

the following linear constraint may be used:

$$x + y \geq 1. \quad (B.12)$$

Assigning the OR relation of two variables

$$z = x \vee y = x + y - xy, \quad (B.13)$$

again contains the nonlinear xy term. Here, a general way to linearize is to substitute $w = xy$ and use (B.7), (B.8), and (B.9) to get the linear constraints

$$z = x + y - w \quad (\text{B.14})$$

$$w \leq x \quad (\text{B.15})$$

$$w \leq y \quad (\text{B.16})$$

$$x + y \leq w + 1. \quad (\text{B.17})$$

However, in the case of the OR relation, the following constraints are well known to realize the same with tighter bounds for the solver (i.e., usually faster to solve):

$$z \geq x \quad (\text{B.18})$$

$$z \geq y \quad (\text{B.19})$$

$$z \leq x + y \quad (\text{B.20})$$

General Boolean expressions

Composite Boolean expressions can be addressed similarly as performed using the substitution above. Consider the following example:

$$z = (w \wedge \bar{x} \wedge \bar{y}) \vee (\bar{w} \wedge x \wedge y). \quad (\text{B.21})$$

By using (B.1), (B.6), and (B.10), it can be rewritten to

$$z = w(1 - x)(1 - y) \vee (1 - w)xy \quad (\text{B.22})$$

$$= w(1 - x)(1 - y) + (1 - w)xy - w(1 - x)(1 - y)(1 - w)xy \quad (\text{B.23})$$

$$= w - wx - wy + xy \quad (\text{B.24})$$

Note that the rule $xx = x$ for $x \in \{0, 1\}$ is used several times. Finally, we substitute $a = wx$, $b = wy$ and $c = xy$ to get the linear constraint

$$z = w - a - b + c, \quad (\text{B.25})$$

and apply (B.7), (B.8), and (B.9) to each expression of a , b , and c to get the corresponding linear constraints.

XOR

A logical exclusive or (XOR) is given by the Boolean relation

$$z = x \oplus y = x\bar{y} + \bar{x}y, \quad (\text{B.26})$$

which can be replaced by

$$z = x(1 - y) + (1 - x)y \quad (\text{B.27})$$

$$= x - xy + y - xy \quad (\text{B.28})$$

$$= x + y - 2xy \quad (\text{B.29})$$

$$= x + y - 2w. \quad (\text{B.30})$$

Now, we could use again (B.15) to (B.16) to express $w = xy$. However, this particular case allows to constrain w to be any integer, and we still get an XOR. The reader may check that there is only one value of w that results in a Boolean result of z , which corresponds to the XOR.

In many cases, the result of the XOR should equal one

$$x + y = 1. \quad (\text{B.31})$$

This is often used to express that either x or y should be true, but not both, for mutually exclusive decision alternatives.

Implication

The Boolean implication $x \rightarrow y$ (x implies y) can be expressed as $x \rightarrow y = \bar{x} \vee y$. Using the method above to state $x \rightarrow y = 1$ would lead to the constraint $1 - x + xy = 1$, for which xy has to be linearized as above. However, the constraint

$$x \leq y \quad (\text{B.32})$$

fulfills the same property, as our reader may check by evaluating the truth table.

Table B.2 Common expressions and their linearization

Nonlinear expression	Linear expression
$X = Y$ if $z = 1$	$Y - M + Mz \leq X \leq Y + M - Mz$
$X = \sum_{i=0}^{w_X-1} 2^i x_i$	$Z = \sum_{i=0}^{w_Y-1} 2^i W_i$
$Z = X \times Y$	$Y - M + Mx_i \leq W_i \leq Y + M - Mx_i$ $-Mx_i \leq W_i \leq Mx_i$

B.3.3 Indicator Constraints

A common need is that a constraint should be only active under a certain condition. Such constraints are called *indicator constraints* and have the form

$$X = Y \text{ if } z = 1 \quad (\text{B.33})$$

where X and Y can be arbitrary types, while z is a Boolean variable: the constraint $X = Y$ should only be active when $z = 1$.

Although not linear, indicator constraints are directly supported by many modern ILP solvers. They can also be rewritten in linear form by using a so-called *big-M* constraint, defined as

$$Y - M + Mz \leq X \leq Y + M - Mz. \quad (\text{B.34})$$

Indeed, for $z = 1$, (B.34) becomes $X = Y$; for $z = 0$, (B.34) becomes $Y - M \leq X \leq Y + M$, which is always true if the constant M is set to a sufficiently large value, i.e., $M \geq |Y - X|$ for any possible value of X and Y : the constraint is disabled.

Indicator constraints are helpful to linearize nonlinear models. An example can be found in Sect. 12.1.4, p. 373.

The use of big-M constraints is typically faster than directly using indicator constraints when supported by the solver. So, big-M constraints should be preferred in terms of performance, but if the M constant is too large, numerical issues may occur. To be more precise, $1/M$ has to be larger than the *integer feasibility tolerance* used in the ILP solver (see Sect. B.4).

B.3.4 Splitting Integers into Their Binary Representation

Another common method to linearize equations is to represent integer variables using binary representation, e.g.,

$$X = \sum_{i=0}^{w_X-1} 2^i x_i, \quad (\text{B.35})$$

with $X \in \mathbb{N}$ and $x_i \in \{0, 1\}$. Another way is to use a one-hot code (an example can be found in Sect. 12.1.4.3 for the different shift values in the constant multiplication). Then, indicator constraints can be used to model the cases depending on the individual bits.

A common example is the linearization of an integer product of two variables:

$$Z = X \times Y \quad (\text{B.36})$$

with $X, Y, Z \in \mathbb{N}$. It can be rewritten as

$$Z = \sum_{i=0}^{w_Y-1} 2^i W_i \quad (\text{B.37})$$

with

$$W_i = \begin{cases} 0 & \text{if } x_i = 0 \\ Y & \text{if } x_i = 1 \end{cases}, \quad (\text{B.38})$$

and x_i the bits of X as defined in (B.35). Equation (B.38) is not linear but can be linearized using a variation of previous big-M constraint:

$$Y - M + Mx_i \leq W_i \leq Y + M - Mx_i \quad (\text{B.39})$$

$$-Mx_i \leq W_i \leq Mx_i. \quad (\text{B.40})$$

B.3.5 Counting Leading and Trailing Zeroes

In [De +17], the objective is to minimize the size in bits of some fixed-point coefficients. To this purpose, a coefficient C is expressed in binary as above:

$$C = \sum_{i=\ell}^m 2^i c_i, \quad (\text{B.41})$$

with $c_i \in \{0, 1\}$, the MSB position m is known, and the constant ℓ is chosen sufficiently small to ensure that the expected coefficient will have some trailing zeroes. To count these trailing zeroes, a set of k boolean variables μ_i is defined as follows:

$$\mu_\ell = 1 \iff c_\ell = 0 \quad (\text{B.42})$$

$$\mu_{\ell-1} = 1 \iff c_\ell = 0 \wedge c_{\ell-1} = 0 \quad (\text{B.43})$$

$$\mu_{\ell-2} = 1 \iff c_\ell = 0 \wedge c_{\ell-1} = 0 \wedge c_{\ell-2} = 0 \quad (\text{B.44})$$

...

$$\mu_{\ell-k} = 1 \iff c_\ell = 0 \wedge c_{\ell-1} = 0 \dots \wedge c_{\ell-k} = 0 \quad (\text{B.45})$$

where k is the maximum expected number of trailing zeroes. The trailing zero count is then simply defined as

$$t = \mu_\ell + \mu_{\ell-1} \dots + \mu_{\ell-k} . \quad (\text{B.46})$$

Of course, the same technique may be used to count leading zeroes.

B.4 Addressing Limitations of ILP Solvers

We finish this overview with two issues that will limit the use of ILP solvers: their run-time on large problem and numerical issues that may jeopardize the solving process.

B.4.1 Run-Time

Since ILP problems are in general NP-hard, an exponential growth of the run-time with the problem sizes should be expected. Small problems are typically solved very fast (well under a second), but there usually exists a problem size “wall” that cannot be surpassed. Near this wall, the time to compute the optimal solution exceeds hours and days. Even patience may not help, as solvers also risk to run out of memory.

Two measures of the problem size are the number of variables and the number of constraints. Here, the experience of the authors is that the number of variables is the more critical. Indeed, adding constraints to a problem, if it reduces the search space (e.g., by excluding values of some variables) can actually dramatically speed up the search, even though the problem has more constraints than before. It usually helps to add constraints that exclude nonoptimal solutions and optimal solutions that can be deduced by problem symmetries.

In addition, good solutions are often found much earlier than a solution that is proven to be optimal. In fact, most of the time of an ILP run is typically spent on proving that the optimal result is actually optimal. Therefore, a common practice is to define a time limit for the tool. Once this limit is reached, the tool either returns the best solution found so far (which is not a proof that this solution is optimal) or answers that no solution was found (which is not a proof that no solution exists). By setting a time limit, we renounce optimality, but ILP remains very useful as a generic heuristic that provides very good (near-optimal) solutions.

Apart from the number variables, the type of the variables also matters. The core solving process of ILP is based on the real-valued LP problem, which is much simpler to solve, thanks to the simplex method [Sch98]. Hence, defining variables as reals wherever possible often helps to reduce the run-time.

It often appears that ILP variables are derived from other integer variables or constants, such that they naturally become integer. For example, a constraint like

$$C = A + B$$

with $A, B \in \mathbb{N}$ can only result in an integer C . Such integer variables may be *relaxed* to real numbers to speed up the optimization. For example, most of the constraints in the ILP problem in Sect. 12.1.4, p. 373, can be relaxed to real numbers.

Another way to reduce the run-time is to provide a heuristic solution (if known) as a “warm-start.” This is in particular helpful for problems which take a long time to find a first feasible solution.

B.4.2 Numerical Instabilities

Another practical limitation of ILP solvers is that they may be sensitive to numerical effects during the optimization. Most solvers use double-precision (binary64) floating-point arithmetic, even when working with binary or integer variables. This entails practical limitations to the numeric range of variables and constants. A binary floating-point number can represent exactly all the integers up to $2^{53} \approx 10^{16}$. However, the main issue is rounding errors in the solving process. To acknowledge it, the typical dynamic range of modern solvers is stated to be in the order of 10^4 [Esp22]. Solvers treat values as integers when the error between a floating-point representation and the nearest integer is below the *integer feasibility tolerance* of the solver, which can be configured by the user. Typical values of this parameter are around 10^{-5} and should not be less than 10^{-9} . Selecting a smaller integer feasibility tolerance might lead additional iterations and thus long run-times. This can easily happen for binary representations of large word sizes as discussed in Sect. B.3.4. Here, a reformulation of constraints can reduce the numeric range.

These numerical aspects should be particularly taken into consideration in the big-M constraints described above. When supported by the tool, indicator constraints should be safer, if slower.

In general, numerical artifact may render the result invalid, in the sense that a solution found by the solver is not a solution to the ILP problem. It is always a good idea to double-check the obtained solutions.

References

- [Ach09] Tobias Achterberg. “SCIP: solving constraint integer programs”. In: *Mathematical Programming Computation* 1.1 (2009), pp. 1–41. URL: <http://link.springer.com/10.1007/s12532-008-0001-1>. (cit. on p. 784).
- [De +17] Davide De Caro, Ettore Napoli, Darjn Esposito, Gerardo Castellano, Nicola Petra, and Antonio GM Strollo. “Minimizing Coeffi-

- lients Wordlength for Piecewise-Polynomial Hardware Function Evaluation With Exact or Faithful Rounding". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 64.5 (2017), pp. 1187–1200. (cit. on p. 794).
- [Esp22] Daniel Espinoza. *Avoiding Numerical Issues in Optimization Models – Online Webinar*. 2022. URL: <https://www.gurobi.com/resource/numerical-issues-webinar/>. (cit. on p. 796).
- [Gurobi] Gurobi Optimization Inc. *Gurobi Website*. 2022. URL: <http://www.gurobi.com>. (cit. on p. 784).
- [IBM22] IBM. *IBM CPLEX Optimizer Website*. 2022. URL: <https://www.ibm.com/de-de/analytics/cplex-optimizer>. (cit. on p. 784).
- [Inc22a] Free Software Foundation Inc. *GNU Linear Programming Kit (GLPK) Website*. 2022. URL: <https://www.gnu.org/software/glpk/>. (cit. on p. 784).
- [Inc22b] Google Inc. *Google Optimization Tools*. 2022. URL: <https://developers.google.com/optimization/>. (cit. on p. 787).
- [MS20] Andrew Makhorin and Heinrich Schuchardt. *Modeling Language GNU MathProg-Language Reference for GLPK Version 5.0*. Tech. rep. Free Software Foundation Inc., 2020. (cit. on p. 786).
- [Sch98] Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley & Sons Ltd, 1998. (cit. on pp. 781, 795).
- [Sit+18] Patrick Sittel, Thomas Schönwälder, Martin Kumm, and Peter Zipf. "ScaLP: A Light-Weighted (MI)LP Library". In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*. 2018, pp. 1–10. (cit. on p. 787).
- [SSK22] Patrick Sittel, Thomas Schönwälder, and Martin Kumm. *ScaLP Project Website*. 2022. URL: <http://unikassel.de/go/scalp>. (cit. on p. 787).
- [SZ15] Gerard Sierksma and Yori Zwols. *Linear and Integer Optimization: Theory and Practice*. 3rd ed. Chapman and Hall (CRC), 2015. (cit. on pp. 781, 786, 787).
- [Tec22] Massachusetts Institute of Technology. *lp_solve Website*. 2022. URL: <https://web.mit.edu/lpsolve/doc/>. (cit. on p. 784).
- [Wil13] H. Paul Williams. *Model Building in Mathematical Programming*. Wiley, 2013. (cit. on p. 787).
- [ZIB22] Zuse Institute Berlin (ZIB). *SCIP – Solving Constraint Integer Programs Website*. 2022. URL: <https://www.scipopt.org>. (cit. on p. 784).

Index

- 2D-norm, 486
3D-norm, 486
4:2 compressor, 181
- AAM, *see* add-add multiplex
absolute error, 67
ac_fixed, 43, 70
accuracy, 66
accuracy in bits, 534
activation function, 712
adaptative logic module, 93, 96, 128, 132
add-add multiplex, 137, 138
adder graph, 367
addition-table-addition, 523
AI, *see* artificial intelligence
ALAP, *see* as late as possible
algebraic function, 357, 486
ALM, *see* adaptative logic module
ALU, *see* arithmetic and logic unit
ANN, *see* artificial neural network
ap_fixed, 43, 70
application-specific arithmetic, 8
application-specific integrated circuit, 9, 15–17, 87, 175, 180, 193, 417, 491, 594, 669
approximate computing, 25
approximation error, 73
arithmetic and logic unit, 311
arithmetic shift right, 309
artificial intelligence, 707, 714
artificial neural network, 707, 717
as late as possible, 177
as soon as possible, 177
ASAP, *see* as soon as possible
ASIC, *see* application-specific integrated circuit
ATA, *see* addition-table-addition
Avižienis number systems, 47
- basic logic element, 87–96, 126–129, 137, 139, 148, 193, 434, 745
bfloat16, 53
BHM, *see* Bull and Horrocks Modified
bias, 49, 338, 343, 348
BIBO, *see* bounded-input, bounded-output
BIBO stability, 675
big-O notation according to Landau, 115, 214, 320
BILP, *see* binary integer linear programming
binade, 49, 58, 68
binary angles, 602
binary integer linear programming, 235
binary tree constant division, 434
binary weight, 152
bipartite method, 503
bit
 position, 34
 weight, 34
BLE, *see* basic logic element
block floating point, 53
Booth encoding, 215
bounded-input, bounded-output, 675, 684, 687
BTCD, *see* binary tree constant division
Bull and Horrocks Modified, 410
- canonical signed digit, 368, 369, 373
carry computation circuit, 138, 139
carry propagate adder, 108, 114, 115
carry recovery, 138, 139
carry-lookahead, 116, 117
carry-save, 110, 131, 154
carry-save adder, 131, 132
carry-save format, 111
carry-select adder, 114, 115
Cascaded-Integrator-Comb, 678

- Cauchy signed-digit decimal system, 46
 causal filter, 673
CCC, *see* carry computation circuit
CIC, *see* Cascaded-Integrator-Comb
CLA, *see* carry-lookahead
CLB, *see* Configurable Logic Block
CMM, *see* constant matrix multiplication
CNN, *see* convolutional neural network
 common subexpression elimination, 369, 410, 740, 749
 compound adder, 124, 333, 347
 compound fast absolute difference, 124
 compound fast adder, 123
 compound triple sum, 124
 compressor, 154
 conditional-sum adder, 116
 Configurable Logic Block, 91
 constant matrix multiplication, 402, 410, 411, 417
 convolution, 674, 716
 convolutional neural network, 708, 709, 715–717, 719, 720, 722, 737, 739, 744, 746–748, 750
 CORDIC, 23, 590, 663
 cosine, 599
 CPA, *see* carry propagate adder
 CR, *see* carry recovery
 CSA, *see* carry-save adder
 CSD, *see* canonical signed digit
 CSE, *see* common subexpression elimination

 DAG, *see* directed acyclic graph
 dark silicon, 3
 data-free quantization, 734
 dataflow, 6
 DCT, *see* discrete cosine transform
 DDS, *see* direct digital synthesis
 deep neural network, 707, 708, 726, 730, 732, 734, 735
 DFQ, *see* data-free quantization
 DFS, *see* depth-first search
 DFT, *see* discrete Fourier transform
 dichotomy-based segmentation, 552
 DiffAG, *see* difference-based adder graph
 difference-based adder graph, 402, 404
 digit-serial arithmetic, 23
 digital signal processing, 88, 94, 95, 98, 214, 222–224, 226, 231–233, 235, 245, 249, 250, 402, 414, 415, 669, 670
 direct digital synthesis, 602
 direct memory access, 742
 directed rounding modes, 71
 discrete cosine transform, 402

 discrete Fourier transform, 7
 divider, 3
 DLFloat, 53
 DMA, *see* direct memory access
 DNN, *see* deep neural network
 dot diagram, 154, 163, 205
 DSP, *see* digital signal processing

 E-method, 486
 E4M3, 53, 729
 E5M2, 53, 729
 EDIF, *see* Electronic Digital Interchange Format
 ELMA, *see* Exact Log-Linear Multiply-Add
 embedded SRAM, 98
 error
 absolute, 67
 relative, 67
 error analysis, 78, 555, 563, 691
 division by quadratic series, 297
 division by reciprocal approx., 289
 exponential, 651
 exponential table-based, 657
 fix-real-KCM, 391
 multipartite, 511
 sine/cosine CORDIC, 609
 sine/cosine multiplier-based, 616
 sine/cosine table-based, 616
 eSRAM, *see* embedded SRAM
 Estrin polynomial evaluation, 561
 Exact Log-Linear Multiply-Add, 727
 exclusive-or, 28, 91, 92, 113, 121, 126, 127, 132
 exponential, 23, 641

 FA, *see* full adder
 faithful operator, 76
 faithful rounding, 76, 487
 fanout, 162
 fast carry chain, 91
 fast carry logic, 91, 125, 148, 182, 184, 231, 276, 278, 288, 315, 338, 385, 580, 604, 613, 635
 fast convolution algorithm, 737
 fast Fourier transform, 366, 386, 402, 469
 FF, *see* flip-flop
 FFT, *see* fast Fourier transform
 field programmable gate array, vi, 4–7, 9, 15–17, 23, 24, 87, 88, 90, 91, 93–96, 98, 126, 128, 129, 132–135, 176, 180, 182, 183, 185, 186, 193, 222–224, 229, 231, 239, 249, 378, 384, 414, 415, 417, 434, 594, 669, 728, 748
 FIFO, *see* first-in first-out

- filter design, 414
finite impulse response, 94, 393, 402, 673
Finite State Machine, 634
FIR, *see* finite impulse response
first-in first-out, 736, 742
fixed-point
 signed format, 41
 unsigned format, 38
fixed_pkg, 43, 70
flip-flop, 87, 89, 91, 93, 94, 96, 98, 135, 137, 139, 177, 179, 180, 189, 193, 194, 749
FloPoCo, *see* Floating Point Core generator, 22
FMA, *see* Fused Mutiply-Add
FP, *see* floating point
FP8, 53, 729
FPGA, *see* field programmable gate array
fpmiminax, 537
FPU, *see* floating-point unit
FR, *see* faithful rounding
fractional part, 39
FSM, *see* Finite State Machine
full adder, 93, 106–109, 112–114, 126, 128, 129, 131, 132, 137–139, 154, 155, 176, 180, 370, 381
function approximation, 12
Fused Mutiply-Add, 4, 324, 347, 469

GA, *see* genetic algorithm
Gappa, 82
Gaussian error linear unit, 713
GCC, *see* GNU compiler collection
GCD, *see* greatest common divisor
GELU, *see* Gaussian error linear unit
generalized parallel counter, 182–184, 187–191, 193, 194
genetic algorithm, 410
GMP, *see* GNU Multiple Precision
GNU compiler collection, 470
GPC, *see* generalized parallel counter
GPGPU, *see* general-purpose GPU
GPP, *see* general-purpose processor
GPU, *see* graphics processing unit
graphics processing unit, 330, 641, 724, 742
greatest common divisor, 250
guard bits, 11, 84, 390, 413, 515, 516, 610, 649

HA, *see* half adder
half adder, 107, 109, 125, 155, 176, 180, 381
half-unit biased, 71–73, 83, 392, 516, 517
Hamming weight, 367
hardware description language, 19, 132

HDL, *see* hardware description language
hexadecimal, 44
hierarchical segmentation, 551
high radix multiplication, 221
high-level synthesis, 43, 70, 453, 454, 457, 470
high-radix
 carry-save, 111, 288, 635
 representation, 45, 384, 445
HLS, *see* high-level synthesis
Horner evaluation, 554
HPC, *see* high-performance computing
HRCS, *see* high-radix carry-save
HUB, *see* half-unit biased

I/O, *see* input/output
IC, *see* integrated circuit
IEEE 754, 48, 69, 339, 345, 350, 353, 356, 357, 469, 626
IIR, *see* infinite impulse response
ILP, *see* integer linear programming
ILSVRC, *see* ImageNet Large Scale Visual Recognition Challenge
ImageNet Large Scale Visual Recognition Challenge, 708
implicit bit, 50
impulse response, 672
inference, 714
infinite impulse response, 393, 402, 673
infinity norm, 534
injection rounding, 346, 351
integer linear programming, 19, 22, 84, 186, 187, 189, 192–194, 214, 234, 240–242, 369, 373, 374, 378, 379, 381, 389, 406, 407, 522, 561, 566, 697–699
International Technology Roadmap for Semiconductors, 2
interpolation, 547
Intersection over Union, 715
inverse cumulative distribution function, 554, 568
IoU, *see* Intersection over Union
ITRS, *see* International Technology Roadmap for Semiconductors

KCM, *see* Ken Chapman's Multiplier
Ken Chapman's Multiplier, 366, 384

LAB, *see* logic array block
last-bit accuracy, 76
LBC, *see* leading bit counter
LDTC, *see* lossless differential table compression
LE, *see* logic element

- leading bit counter, 316
- leading one counter, 316, 548
- leading one detector, 316
- leading zero counter, 316, 338, 341, 345, 350, 356, 548
- least significant bit, 26–28, 36, 38, 39, 41, 42, 68, 109, 110, 170, 176, 177, 194, 206–208, 215, 227, 242, 262, 313, 368, 381, 386, 444, 458, 487, 610, 626, 649, 662, 663, 690, 699
- limit cycle oscillations, 675, 688
- linear time-invariant, 672
- LNS, *see* Logarithm Number System
- LOC, *see* leading one counter
- LOD, *see* leading one detector
- logarithm, 23, 662
- Logarithm Number System, 23, 56, 59, 329, 478, 724, 727, 730
- logic array block, 93, 96, 128
- long short-term memory, 709
- look-up table, 46, 89–94, 96, 98, 126–129, 132, 133, 138, 182–184, 189, 214, 224, 228–231, 276, 314, 366, 384, 407, 417, 430, 434, 491, 594, 728, 746, 748, 749
- lossless differential table compression, 497–499, 502
- LSB, *see* least significant bit
- LSTM, *see* long short-term memory
- LTI, *see* linear time-invariant
- LUT, *see* look-up table
- LZC, *see* leading zero counter
- MAC, *see* multiply-accumulate
- machine-efficient polynomial approximation, 536
- MAE, *see* mean absolute error
- MAG, *see* minimized adder graph
- mAP, *see* mean Average Precision
- MCM, *see* multiple constant multiplication
- mean absolute error, 731
- mean Average Precision, 715
- mean squared error, 731
- memory LAB, 96
- merged arithmetic, 152, 157
- MILP, *see* mixed integer linear programming
- MIMO, *see* multiple input, multiple output
- minimax polynomial approximation, 533
- minimized adder graph, 370
- minimum signed digit, 368, 369, 373
- minimum spanning tree, 410
- MLAB, *see* memory LAB
- MLP, *see* multilayer perceptron
- most significant bit, 26–28, 36, 37, 39, 42, 95, 106, 146, 162, 174, 176, 177, 194, 206–208, 221, 228, 242, 262, 267, 313, 314, 323, 381, 386, 516, 626, 690
- MPFR, *see* GNU Multiple Precision
- Floating-point with correct Rounding
- MSB, *see* most significant bit
- MSD, *see* minimum signed digit
- MSE, *see* mean squared error
- MST, *see* minimum spanning tree
- multilayer perceptron, 709
- multipartite method, 503
- multiple constant multiplication, 373, 377, 402–408, 410, 417, 678, 679, 697–699, 702, 735
- multiple input, multiple output, 671, 690
- multiplexer, 89–92, 113, 116, 126, 127, 138, 139, 417, 745
- multiply-accumulate, 94, 95, 157, 742, 748
- MUX, *see* multiplexer
- n-dimensional reduced adder graph, 402
- NaN, *see* Not a Number
- negation, 37
- Newton-Raphson, 291, 303
- NOF, *see* non-output fundamental
- non-output fundamental, 403
- normal floating-point number, 49
- normalization, 323, 636
- normalizer, 548, 550
- Not a Number, 51, 331, 343
- on-the-fly conversion, 281, 587, 590
- online arithmetic, 48
- operator fusion, 10
- operator specialization, 9
- overflow, 43, 71
- P-D diagram, 273, 582
- PAG, *see* pipelined adder graph
- parallel polynomial evaluation, 558
- parallel-prefix operation, 321
- PE, *see* Processing Element
- pipeline, 6
- pipelined adder graph, 406
- pipelined MCM, 404–406, 410
- posit number format, 53
- positional number system, 46
- post-training quantization, 725, 734
- precision
 - of a fixed-point format, 39

- prefix operation, 118, 321
prescaling, 283
probit, 554, 568
PTQ, *see* post-training quantization
- Q, 43
QAT, *see* quantization-aware training
quantization, 43
quantization step, *see* unit in the last place
quantization-aware training, 726, 727, 730, 733, 734, 748
- RA, *see* reduced area
radian angles, 599
RAG-n, *see* n-dimensional reduced adder graph
RAM, *see* random access memory
random access memory, 96, 98
range
 of a fixed-point format, 39
 of a function, 480, 602
rational function, 486
RCA, *see* ripple-carry adder
RCCM, *see* reconfigurable constant coefficient multiplier, 417, 744–746
rectangular multipliers, 204
rectified linear unit, 712, 713
recurrent neural network, 709
Red, Green, Blue, 711, 714
reduced area, 177–180, 193
reduced pipelined adder graph, 405, 410, 415, 749
redundancy factor, 272, 581
redundant number systems, 47
relative error, 67
ReLU, *see* rectified linear unit
Remez, 533
Residue Number System, 23, 211
RGB, *see* Red, Green, Blue
ripple-carry adder, 108, 113, 114, 126, 128, 129, 131–133, 135, 138, 139, 194, 222, 380, 745
Rivest–Shamir–Adleman, 104
RNN, *see* recurrent neural network
RNS, *see* Residue Number System, 211
ROM, *see* read-only memory
round to nearest, 68, 487
 ties to away, 69, 344
 ties to even, 69
 ties to up, 261
round toward infinity, 71
round toward zero, 71
round up, 71
rounding, 43
rounding bit, 70, 338, 344, 349, 634, 637
rounding tie, 69
RPAG, *see* reduced pipelined adder graph
RSA, *see* Rivest–Shamir–Adleman
- SAD, *see* sum of absolute difference
saturated arithmetic, 71, 105
scaling factor, 40
SCM, *see* single constant multiplication
SD, *see* signed digit
SFG, *see* signal flow graph
sfixed, 41
SGD, *see* stochastic gradient descent
shift register LUT, 96, 137
sign bit, 36, 37, 41, 49
sign extension, 42, 162, 309
sign-magnitude, 37
signal flow graph, 407, 408, 410, 411
signed digit, 368, 410
SIMD, *see* single instruction, multiple data
sine, 599
single constant multiplication, 369, 370, 373, 374, 377, 381, 401, 402, 405
single input, single output, 678
single instruction, multiple data, 6
SISO, *see* single input, single output
SMT, *see* Symmetric Multi-Threading
SoC, *see* System-on-Chip
Sollya, 531
SOP, *see* sum of products
SOPC, *see* sum of products by constants
square root, 553, 574
SRAM, *see* static RAM
SRL, *see* shift register LUT
SRT division, 272, 278
static RAM, 87, 89–91, 96
STE, *see* straight-through estimator
sticky bit, 314, 335, 350, 458, 638
STL, *see* standard template library
stochastic gradient descent, 714, 731, 732
straight-through estimator, 733, 734
subnormal number, 50
sum of absolute difference, 134
sum of products, 736, 737
sum of products by constants, 395, 402, 407, 408, 410, 412, 414, 677, 679, 691, 692, 750
SVG, *see* Scalable Vector Graphics
switched ripple-carry adder, 113
Symmetric Multi-Threading, 6
System-on-Chip, 132

- table-and-addition method, 503
TableMaker’s Dilemma, 76
Taylor approximation, 531
tensor processing unit, 735, 741
ternary adder, 131
ternary balanced notation, 368
TF32, 53
tie, 69
TPU, *see* tensor processing unit
transpose, 407, 678
trigonometric identities, 600
truncated
 bit heap, 167
 multiplier, 209, 211, 237
 squarer, 446
truncation, 70
two’s complement, 36, 41
 negation, 37
 representation, 37, 41
ufix, 38
ulp, *see* unit in the last place
uniform segmentation, 541
unit in the last place, 28, 39, 67, 68, 168,
 209, 261, 386, 393, 576, 610, 651, 688
unsigned, 34, 39
very high speed integrated circuit
 hardware description language,
 132
very large scale integrated circuit, 121, 132,
 135
VGG, *see* Visual Geometry Group
VHDL, *see* very high speed integrated
 circuit hardware description
 language
Visual Geometry Group, 722
VLSI, *see* very large scale integrated
 circuit

WCPG, *see* Worst-Case Peak-Gain
weight, 34, 152
Winograd fast convolution, 737
WLO, *see* Word Length Optimization
Word Length Optimization, 25
Worst-Case Peak-Gain, 689

XOR, *see* exclusive-or

z-transform, 683
Zuse Z3, 114